

# **For Reference**

---

**NOT TO BE TAKEN FROM THIS ROOM**

Ex LIBRIS  
UNIVERSITATIS  
ALBERTAENSIS









THE UNIVERSITY OF ALBERTA

AN APL COMPILER-INTERPRETER

by



Dennis A. James

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

Fall, 1972



Thesis  
72F-106

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled AN APL COMPILER-INTERPRETER, submitted by Dennis A. James, in partial fulfillment of the requirement for the degree of Master of Science.

Date May 26 1972.....



## ABSTRACT.

Present implementations of APL interpret APL expressions in an environment-insensitive manner. Operators are applied directly to their operands creating intermediate results for the next operation. This method of interpretation can be very inefficient in terms of storage and central processor requirements. APL expressions containing scalar APL arithmetic operators, inner and outer products, and selection expressions can be reduced to a set of simpler primitive functions which minimize the creation of intermediate results. Evaluation of these reduced expressions can be shown to be more efficient, in array operations, than present methods of evaluation. A system has been developed and a sub-set of that system has been simulated which performs those reductions on APL expressions. This simulation is composed of three units, or machines, which compile object modules, interpret them, and perform the required evaluations. The interpretive phase is done in an environment-sensitive manner. The simulation is called APLM.



Digitized by the Internet Archive  
in 2020 with funding from  
University of Alberta Libraries

<https://archive.org/details/James1972>

## TABLE OF CONTENTS.

CHAPTER		PAGE
One.	Introduction.	1
	1.1 APL.	1
	1.2 The Problem.	2
	1.3 A Solution.	2
	1.4 General Outline.	5
Two.	Mathematical Transformations.	6
	2.1 Drag-along and Beating.	6
	2.2 Example and Definitions.	8
	2.3 Storage Access Function.	9
	2.4 Selection by Beating.	11
Three.	General Objects in APLM.	16
	3.1 Memory Arrangement.	17
	3.2 Memory Management.	19
	3.3 APLM Registers.	20
	3.3.1 Name Index Table.	20
	3.3.2 Name Table.	21
	3.3.3 Location Stack.	23
	3.3.4 Iteration Control Stack.	25
	3.3.5 Value Stack.	27
	3.3.6 Instruction Stack.	28
	3.3.7 Addition Registers.	31





3.4	Data Structures.	32
3.4.1	Scalars.	32
3.4.2	Arrays.	32
3.5	Program Segments.	34
Four.	C-Machine.	36
4.1	CM Object Modules.	36
4.2	API Overview.	37
4.3	Object Module Instructions.	42
4.3.1	Storage Management.	42
4.3.2	Control Instructions.	44
4.3.3	Dyadic Scalar Operators.	45
4.3.4	Monadic Scalar Operators.	47
4.3.5	Selection Operators.	48
4.3.6	Immediate Evaluation Operators.	48
4.3.7	Deferrable Operators.	49
4.3.8	Ccompound Operators.	49
4.4	Instruction Formats.	50
4.5	Monadic and Dyadic Opcodes.	56
Five.	D-Machine.	57
5.1	Introduction.	57
5.2	DM Object Instructions.	57
5.2.1	Simple Instructions.	58
5.2.2	Control Instructions.	59
5.2.3	Micro-instructions.	61
5.3	Interpretation of DM Instructions.	64



	5.3.1	Storage Management Instructions.	65
	5.3.2	Control Instructions.	67
	5.3.3	Scalar Operators.	70
	5.3.4	Selection Operators.	71
	5.3.5	Non-deferrable Operators.	71
	5.3.6	Subscript Operator.	73
	5.3.7	General Dyadic Forms.	77
	5.3.8	Reduction Operator.	78
	5.3.9	Ranking Operator.	80
	5.3.10	Rotation Operator.	83
	5.3.11	Membership Operator.	83
	5.3.12	Compress and Expand.	85
Six.		The E-Machine.	88
	6.1	Examples.	88
	6.2	Indexing Non-subscripted Variables.	93
	6.3	Indexing Subscripted Variables.	96
	6.4	Control Transfer.	101
Seven.		Sample APL Expression.	103
	7.1	Compilation by CM.	103
	7.2	Interpretation by DM.	105
Eight.		APLM Simulation.	116
	8.1	Monitor and Initializers.	116
	8.2	Maincycle.	117
	8.3	D-Machine.	117
	8.4	Load Instructions.	118



8.5	Control and Storage Access.	118
8.6	Scalar Arithmetic Operators.	119
8.7	Select Operators.	119
8.8	Program Constants.	120
Nine.	Conclusions.	128
9.1	Summary.	128
9.2	Extensions and Refinements.	130
Appendix A.		
	Monitor and Initializers.	132
Appendix B.		
	Maincycle Listing.	141
Appendix C.		
	Dmachine Listing.	144
Appendix D.		
	Load Listing.	149
Appendix E.		
	Control and Storage Access.	152
Appendix F.		
	Scalar Arithmetic Operators.	155
Appendix G.		
	Selection Operators.	162
Appendix H.		
	Program Constants.	165



## ILLUSTRATIONS.

NAME	TITLE	PAGE
Figure 3.1	Structure of Memory.	18
APLM Stacks.		23-28
EM Instruction Formats.		29-31
Figure 3.2	Array Descriptor.	33
Figure 3.3	Segment Descriptor.	35
Figure 4.1	APL Scalar Functions.	38
Figure 4.2	APL Inner Products.	39
Figure 4.3	APL Outer Products.	39
Figure 4.4	APL Mixed Functions.	40
Figure 4.5	Notes to Figure 4.4.	41
DM Instruction Formats.		50-55
Figure 8.1	DMACCODE Flow-chart.	121
Figure 8.2	MAINCYCL Flow-chart.	122
Figure 8.3	DMACHINE Flow-chart.	123
Figure 8.4	MDADIC Flow-chart.	124
Figure 8.5	GDF Flow-chart.	125
Figure 8.6	ASGN Flow-chart.	126
Figure 8.7	SELECT Flow-chart.	127





## INTRODUCTION

### Chapter One

An indicator of the usefulness of computers is the abundance of programming languages varying from machine level languages to high level languages such as APL. However, there must exist a process, for each language, by which programs written in that language may be translated and expressed in the machine level language of the computer. As languages are designed to make the computer more accessible to the wide range of problem-solvers, the interface between the programming language and the machine language becomes more complex.

#### 1.1 APL.

APL is a precise mathematical notation language, which may also be used as an array-oriented programming language. Therefore, the problem-solver can program a computer in the same notation that he expresses the problem. The primitive functions in APL are complex, and do not generally fit well onto many computer systems. For example, most computer systems are element-oriented, whereas APL is array-oriented, hence there is a mismatch between the programming language and the system. However, APL has been implemented by Abrams



(1966), Breed, Lathwell et al. (1968), and reference manuals written by Berry (1968), and Pakin (1968).

## 1.2 The Problem.

Programs written in APL can be regarded as descriptions of the results as well as a method of obtaining them. It is therefore reasonable to analyze the environment of APL operators before applying them to their operands. Present implementations of APL interpret programs in an environment-insensitive process. That is, each operator is applied to its immediate operands without analyzing the overall context of that operator in the APL expression. Unnecessary evaluations and storage of unnecessary intermediate results occur in environment-insensitive interpretive implementations. The problem therefore is to develop an environment-sensitive interpretive system in which the evaluations made in APL programs are minimized.

## 1.3 A Solution.

The mathematical properties of some APL operators have been analyzed and transformations have been developed by which certain classes of APL expressions can be reduced to a simpler set of primitive operations. F. S. Abrams (1970) has shown that application of those transformations and



execution of the resulting expression is more efficient, in terms of storage and central processor unit requirements, than present implementations of APL. In his dissertation, Abrams develops and verifies transformations which may be applied to selection and reduction expressions, inner and outer products, and any combination thereof, containing one inner or outer product. Selection expressions are those APL expressions consisting of a sequence of the operators take, drop, reversal and transpose. Reduction expressions are APL expressions which contain scalar arithmetic operators applied to their operands by the reduction operator. These transformations are implemented by changing the procedure by which elements of the operands involved are accessed. For example, transposition of a matrix is facilitated by accessing the elements of that matrix in column-major order rather than creating a temporary copy of the transposed matrix. Abrams outlines a system which applies these transformations to programs written in APL.

Programs and arrays are permanently stored in the system. Interpretation of the programs occurs in a two-phase process. The first phase analyzes the environment of APL operators and their operands in an expression and generates a series of instructions from which the second phase performs the operations. Storage is allocated and de-





allocated dynamically as directed by the first phase. Abrams developed the theory of the system and outlined a possible implementation.

The object of this thesis is to implement a sub-set of Abrams's system, by simulation on an IBM 360/67 system. The simulation consists of three phases, or machines, which compile object code, analyze the environment at execution time, and perform the required evaluations. Stacks form the storage structures upon which the system functions. Instructions are stacked, and analyzed when the system encounters selection, reduction or outer product operations. The process of stacking the instructions is called deferral. The analysis phase determines the environment of the operation at execution time by examining the stack of instructions, and initiates execution by transferring control to the execution phase. Storage is allocated and transformations are effected by the analysis phase. Extensions have been added to the system by which outer products may be transformed more efficiently. Outer products may be transformed if the right-hand operand is an array-valued expression as well as an array, as Abrams defined. The simulation has been designed to include only those APL operators whose interpretation is aided by the processes of drag-along and beating.





#### 1.4 General Outline.

Chapter Two briefly outlines the transformations which may be applied to APL expressions, and gives examples of their application. Chapter Three discusses the logical units of the implementation such as stacks, registers and memory. Chapters Four, Five, and Six discuss the purposes and functions of the processes used to translate APL operators into a series of machine executable instructions. Chapter Seven traces an APL expression as it is interpreted by the machine. Chapter Eight discusses the programs written to implement AFIM. Listings of the programs are contained in the appendices.



# Mathematical Transformations

## Chapter Two

### 2.1 Drag-along and Beating.

Present implementations of APL interpret each operator by evaluating its operands to temporary space. The arrays in temporary storage are then used as operands for other operators which in turn are evaluated to temporary storage. This process may be very inefficient in terms of storage requirements and in turn, generates many extra stores and fetches of array elements. Interpretation of the following expression illustrates this inefficiency.  $4 \uparrow (A + B \times C) * 2$

A literal interpretation of this expression would result in the following execution sequence.

- A. The elements of the array C would be negated and stored as an intermediate result.
- B. The corresponding elements of B and the intermediate result would be multiplied and stored as a second intermediate result.
- C. The corresponding elements of A and the second intermediate result would be added and stored as the third intermediate result.



- D. Each element of the third intermediate result would then be squared and stored as the fourth intermediate result.
- E. The first four elements of the fourth intermediate result would then be stored as the fifth intermediate result.

If A, B, and C were 20 element vectors, 80 words of memory would be required for intermediate results, and 80 stores and 80 fetches would be made to and from temporary results.

The context of the expression indicates that it is reasonable to access only the first 4 elements and perform the required operations on them. Only 16 words of intermediate results, 16 fetches, and 16 stores would then be required. It would also seem advantageous to generate each element of the final result element-by-element, thus eliminating all temporary results, and their resulting stores and fetches.

The process of environment-sensitive evaluation is embodied in two complementary processes, drag-along and beating, which are the theoretical basis of APLM. In the following sections, some interesting properties of drag-along and beating are informally illustrated by means of selected examples.



## 2.2 Example and Definitions.

Stack instructions for the APL expression in the previous section could appear as follows, if execution were deferred until the take instruction is encountered.

ICAD	C
MINUS	
ICAD	B
MUITIPLY	
ICAD	A
AED	
ICAD	2
POWER	
IDS	4

The take operator denctes a selection of elements of the operand as the resulting array. Notice that the following series of stack instructions would produce the same result.

ICAD	4 TAKE C
MINUS	
ICAD	4 TAKE B
MUITIPLY	
ICAD	4 TAKE A





ALD

ICAD

2

FCWER

A similar APL expression would be  $((4 \uparrow A) + (4 \uparrow B) \times - (4 \uparrow C)) \times 2$

That is, instead of accessing all the elements of the arrays, only the required elements are accessed, eliminating unnecessary evaluations. The process of deferring execution of stack instructions is called drag-along. The process of changing stack instructions is called beating.

### 2.3 Storage Access Function.

This section explains the procedure used for accessing a required element of an array. For example, assume that the element  $A[;L]$  is to be accessed, where  $[;L]$  means

$[L[1];L[2];L[3];L[4];\dots \dots;L[N]]$

and each element of  $A$  occupies one word in memory. This element is located at  $VBASE + (\rho A) \downarrow L$ , where  $VBASE$  is the address of  $A[0;0;0;\dots \dots;0;0]$ .

Thus, subscripts of arrays stored in row-major order are representations of numbers in a mixed-radix number system (Knuth (1968) p. 297). This method of accessing an element does not favor any particular array coordinate. The method of accessing an element may be re-written as

$$VBASE + ABASE + \uparrow / DEL \times L$$



Where ABASE is an additive displacement from VBASE. DEL is a weighting vector, which has one element for each coordinate of the required array, and is computed as follows;

$$DEL[N] \leftarrow 1$$

$$DEL[I] \leftarrow DEL[I+1] \times (\rho A)[I+1]$$

$$I \in (1, N-1)$$

DEL is used to calculate the base for the decode operator used in

$$VBASE + (\rho A) \perp L.$$

This function is known as the storage access function.

An array descriptor, DA, contains the weighting constant vector, VBASE, and ABASE, for the array it describes. DAs are fully described in Chapter Three. Thus the DA for each array contains all the information required by the storage access function to access any element of that array. The following sections explain how the selection operators (take, drop, reversal, transpose), may be evaluated by the AFLM by changing the information in the DA which describes that array. These evaluations are the AFLM equivalent of heating.



## 2.4 Selection by Beating.

Recall that the DA for array M contains the following information.

RANK	$\rho \rho M.$
RVEC	$\rho M.$
VEASE	Location of first element of ,M.
ABASE	Constant term of access function.
DEL	Vector of weighting coeffic- ients of the storage access function.

Let A and M be arrays conformable for the operation  $A \uparrow M$ . The following changes to constants in the DA for M facilitate the take operation.

$$ABASE \leftarrow ABASE + DEL + . \times (A < 0) \times RVEC - |A$$

$$RVEC \leftarrow |A$$

That is, if the array is accessed using the transformed DA, its elements will be accessed in the manner similar to that of the array  $A \uparrow M$ . Therefore, the elements of the array need not be repeated or re-organized to represent the selection operation. If it is required that the array M remain in the system intact, its DA is copied to a new DA in



memory, and the transformations are applied to the copied DA. Thus, when the array M is to be accessed, the former DA is used to access its elements. However, if the array  $A \uparrow M$  is to be accessed, the transformed DA is used. The transformation of the DA is known as the machine process of heating.

The transformations applied to the DA of M to effect the operation  $A \uparrow M$  are as follows.

$$ABASE \leftarrow ABASE + DEL + . \times (A > 0) \times |A|$$

$$RVEC \leftarrow RVEC - |A|$$

The transformations applied to the DA of M to effect the operation  $\phi[J]M$  are as follows.

$$ABASE \leftarrow ABASE + DEL[J] \times RVEC[J] - 1$$

$$DEL[J] \leftarrow -DEL[J]$$

The transformations applied to the DA of M to effect the operation  $A \oslash M$  are as follows.

$$R \leftarrow RVEC$$

$$D \leftarrow DEL$$

$$RANK \leftarrow 1 + \lceil |A| \rceil$$

$$L \leftarrow 1$$

$$DEL \leftarrow RANK \uparrow DEL$$

$$RVEC \leftarrow RANK \uparrow RVEC$$

$$REPEAT; RVEC[I] \leftarrow \lfloor |A| / (I - A) \rfloor / R$$

$$DEL \leftarrow \lceil |A| / (I - A) \rceil / R$$

$$I \leftarrow I + 1$$

$$\rightarrow (I \leq RANK) / REPEAT$$





The transformations applied to the DA of M to effect subscripting by a scalar J, along coordinate K, are as follows.

$$ABASE \leftarrow ABASE + DEL[K] + J$$

$$DEL \leftarrow (K \neq 1 \text{ RANK}) / DEL$$

$$RVEC \leftarrow (K \neq 1 \text{ RANK}) / RVEC$$

$$RANK \leftarrow RANK - 1$$

Vectors of consecutively increasing or decreasing elements are encoded as j-vector (len,org,s). The number of elements in the vector is called len and the smallest element is called org. The first element is org and the following elements increase in value monatomically if the value of S is 1. The elements decrease monatomically to org if the value of S is 0. The transformations applied to the DA of M to effect subscripting by a j-vector, along coordinate K, are as follows.

$$ABASE \leftarrow ABASE + DEL[K] \times ORG + (LEN - 1)$$

$$RVEC[K] \leftarrow LEN$$

$$DEL[K] \leftarrow -DEL[K]$$

$$END;$$



For example, consider an expression similar to that in the first section of this chapter.

The stack instructions would appear as follows, before encountering the take instruction.

LCAD	C
MINUS	
LCAD	B
MULTIPLY	
LCAD	A
AED	
LCAD	2
PCWER	
LCAD	D

Let A, B, and C be ten by ten matrices and let the vector D be 2 4. The DAS for A, B, and C are identical except for the VBASE values. ABASE is 0, RANK is 2 and RVEC is 10 10. The vector DEI is 10 1. According to the transformations stated earlier in this section, the take instruction can be effected by beating the DAS of the operands. Therefore, the AFLM would apply the TAKE transformations to the DAS of A, B, and C, as outlined earlier in this section. The vector A in the transformation



is the take operand 2 4. ABASE becomes 8 6 and FVEC becomes 2 4 and these values are placed in their respective fields in the DAs for A, B, and C. Evaluation of the take operator is then complete because every time the arrays A, B, and C are accessed through the transformed DAs, they would appear to be the result of the take operation. Any further evaluations would be carried out only on the required elements of A, B, and C. A similar process is applied to the stack instructions when any selection operator is evaluated.

The remaining chapters of this thesis describe a system which applies the processes of drag-along and beating to APL programs in an interpretive mode.



## General Objects of APLM

### Chapter Three

APLM is divided into three logical units or machines, the C-Machine, the D-Machine, and the E-Machine. APLM is principally a stack machine which accepts as source code, programs written in APL. An APL program is stored in memory as one or more program segments. Arrays of data are stored in linearized row-major order. Each program segment has a corresponding segment descriptor which contains location and size information about its program segment. Similarly, each array of data is referenced by at least one array descriptor which contains information for accessing that array. Each segment descriptor has a corresponding entry in the Name Table (see section 3.3). The C-Machine (convert) compiles a series of D-machine instructions from APL source code. The D-Machine (deferral) interprets these instructions and compiles a series of E-Machine (execution) instructions in a stack. The E-Machine performs the required calculations and transformations on the arrays of data, as directed by the instructions in the stack. A compilation of APL source code by the CM (C-Machine) can be used as an object module and need not be re-compiled each time it is to be executed. The D-Machine and the E-Machine provide no object modules. Data



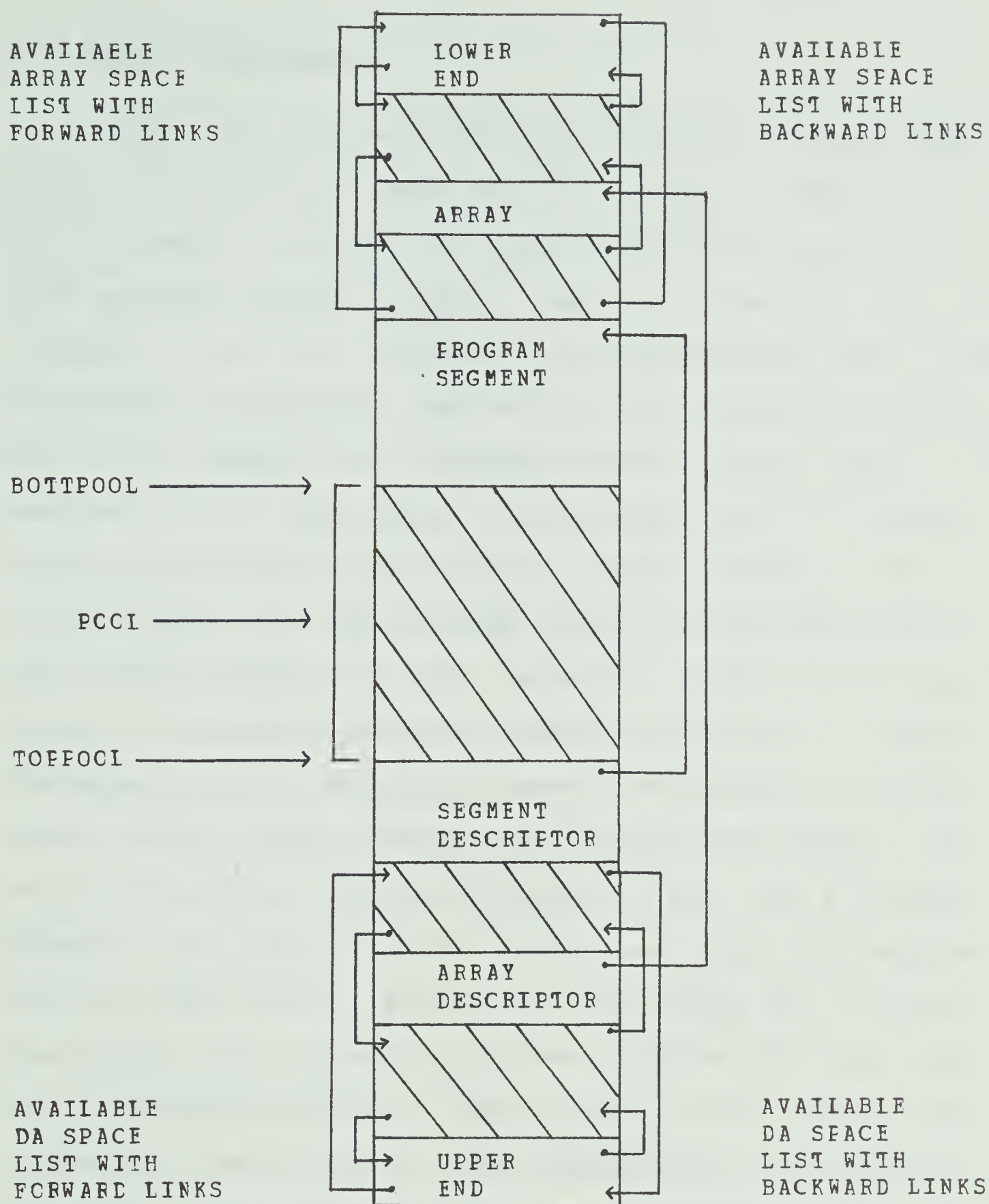


arrays and their descriptors are discussed in section 3.4. Program segments and their descriptors are discussed in section 3.5. Figure 3.1 illustrates the structure of M.

### 3.1 Memory Arrangement.

Memory contains data arrays, programs segments, and their descriptors. Program segments and data arrays are stored from low-order memory (M) toward high-order memory. Descriptors are stored from high-order M toward low-order M. The available memory between the descriptors and their objects is called FCOI. Another portion of AFIM memory contains two tables, the Name Index Table and the Name Table. These tables provide indirect addressing for the names and the descriptors of segments and arrays. All entries in M have a two-word prefix which contains a reference count (RC), (see Collins (1965)), a length value, and a filler count. The length value is the length of the entry and the filler count is the length of the unused portion of the entry, in the case that a larger entry had previously occupied it.





Structure of Memory

Figure 3.1



### 3.2 Memory Management.

Space for new descriptors is allocated from the high-order end of PCOL and space for new program segments and data arrays is allocated from the low-order end of PCCL. When an entry becomes vacant, that is, when its RC is reduced to 0, it is linked in an availability list. The descriptor-availability list and the array-availability list are both forward and backward-linked. When space is required, the appropriate availability list is searched using the first-fit method (Knuth (1968) 436,ff). If no vacant entry is large enough, space is allocated from the appropriate boundary of PCCL. However, if PCOL is not large enough, a garbage-collection process is initiated. Several characteristics of descriptors make it worthwhile to collect vacant array spaces before vacant descriptor spaces. Many of the descriptor's associated arrays local to a program segment are likely to be of the same size and therefore could be used intact. Because of drag-along and beating, descriptors are expected to have a shorter life-time than their associated objects. Since arrays and segments are larger and more stable than descriptors, collection of available array spaces should be more economical than collection of available descriptor spaces.

When available array space is to be collected, array



descriptors are scanned and a linked list is set up which ties together all descriptors referring to the same entry. Arrays are then compacted toward low-order M with adjustments to their pointers in the referent descriptors. Available spaces adjacent to FOOL are merged with FOOL.

When available array descriptor space is to be collected, an algorithm similar to that for array spaces is used. As each descriptor is compacted toward high-order M, the pointer in the Name Table is changed appropriately. Available descriptor space is collected only if the collection of array space does not free the required amount of contiguous memory.

### 3.3 AFLM Registers.

Six register-like structures in AFLM provide the mechanisms for memory accesses, program control, and processing of data.

#### 3.3.1 Name Index Table. (NIT)

NIT is generated by the C-Machine (CM), and is used only by the CM. As the CM generates an object module from an APL program, it must use indirect addressing because the environment of the system at execution time is unknown.







This is accomplished by assigning each named object in an APL program a number, which is called INX. Named objects are referenced in the DM and EM by their unique INX values. NIT is a list of the names of named objects and their corresponding INX values. Each time the CM encounters a named object, it searches NIT for that name. If the name is not found, a new entry is generated in NIT. In either case the corresponding INX value is used in place of the name from that point on.

### 3.3.2 Name Table. (NT)

NT is generated by the CM and the DM and used only by the DM. The CM generates an NT entry each time it generates a program segment, and each time a data array is initialized. The corresponding NT entries basically consist of the INX value of the named object and a pointer pointing to the segment descriptor or the array descriptor for that object. When that object is referenced by its INX value, NT is searched associatively using the INX value as the key. The pointer in the corresponding NT entry indirectly indicates the location of the program segment or the data array.

The DM generates entries each time a program segment is to be interpreted. An NT entry is generated for each



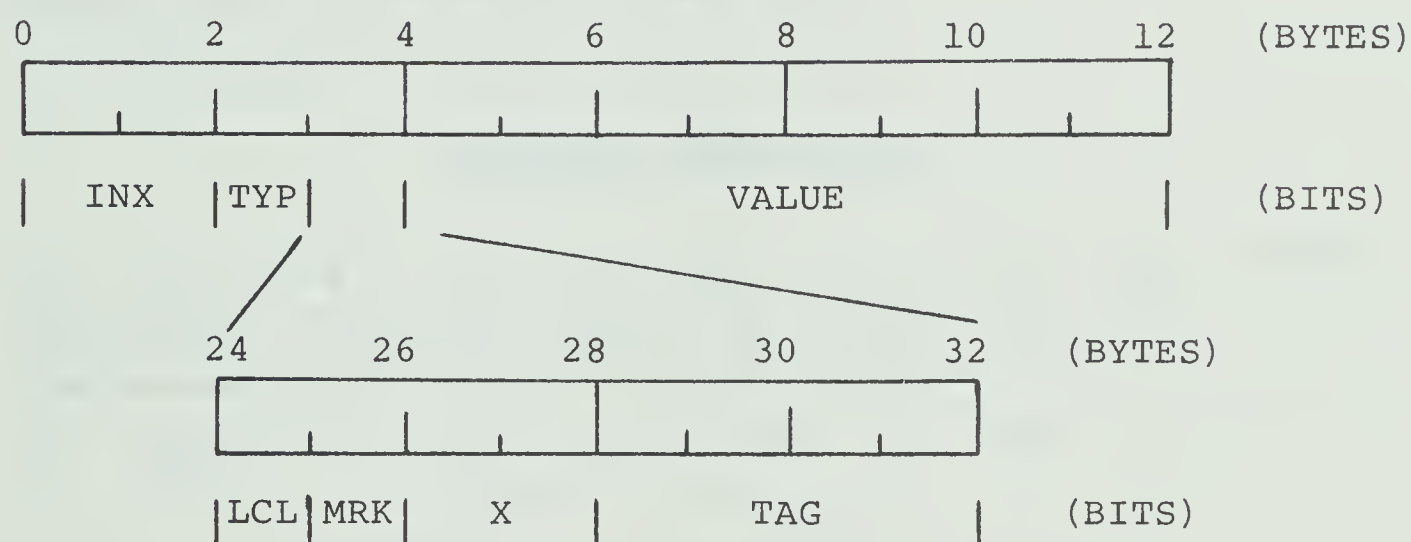
parameter, local variable and the result, of a program segment. When NT is searched associatively, from the top, all references to named objects will be made through the NT entries generated for that program segment, unless they are global variables. NT entries for global variables occur in the portion of the NT created by the CM, or do not exist at all. If an NT entry for a variable cannot be found, a new entry is generated for the new global variable. This method of accessing named objects facilitates the functions of local and global variables as required by the syntax of API. When the program segment is terminated, all NT entries local to that program segment are popped.

A detailed description of an NT entry follows. The first two bytes of the twelve-byte entry contain the INX value of the named object. Byte three contains a type flag (TYPE), which describes the type of data referenced. Permissible types in the TYPE field are boolean, integer, character, real, and long-real (double-precision). They are encoded as BOCL, INT, CHAR, REAL, and IREAL respectively. Bit twenty-four contains an ICL flag which has the value 1 for a local variable and 0 for a global variable. Bit twenty-five contains an MRK flag, which contains the value 1 for the first local variable for the program segment, and 0 for the other local variables. The fourth byte contains a TAG flag which identifies the object referenced by the NT



entry. Permissible identifiers in the TAG field are; ST for scalar quantities, JT for encoded j-vectors, UT for as-yet undefined identifiers, DT for arrays, and FT for program segments. Bytes five through twelve are labelled VALUE. The VALUE field contains the actual values for entries with TAG values of JT or ST, otherwise it contains a pointer to a program segment or an array descriptor. All other fields are unused and labelled X.

### Name Table Entry



### 3.3.3 Location Stack. (IS)

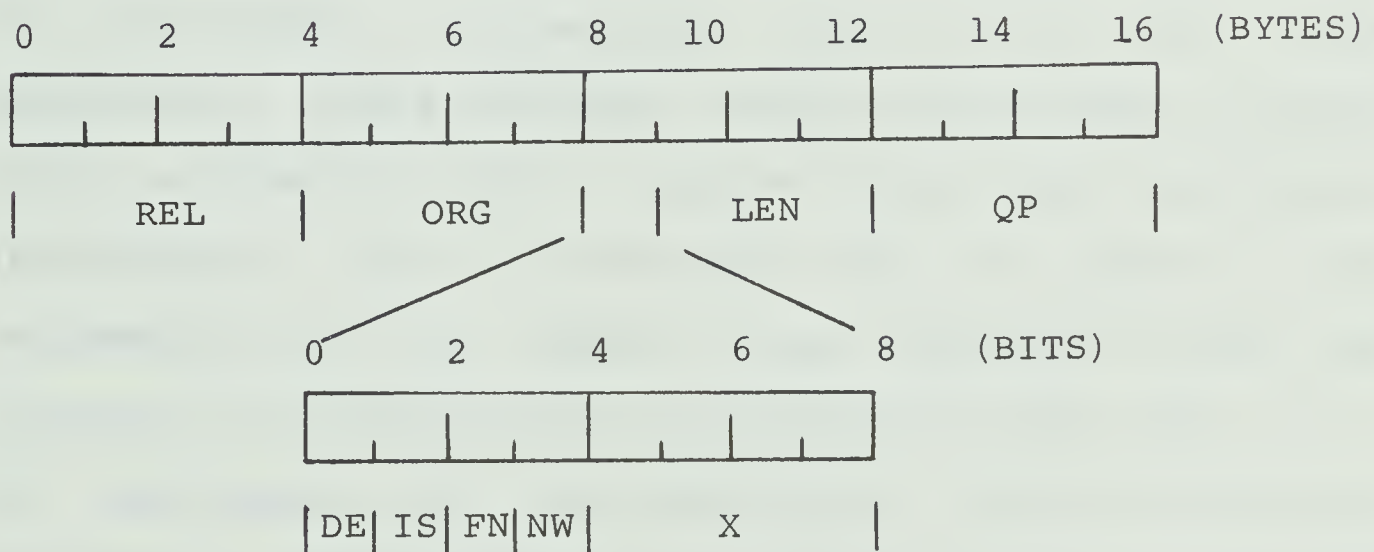
IS contains information about the current program segment with respect to instruction accessing. Program segments are characterized by a starting address and a





length. Each IS entry contains the starting address (ORG), the length (LEN), a counter (REL), an Instruction Stack pointer (QP), and control information. The control information consists of 5 flag bits, DE, IS, FN and NW. DE informs the system whether the DM or the EM should be given control. IS has the value 1 if this segment is associated with an iteration, otherwise it has the value 0. FN has the value 1 if this is the main segment of an active function, otherwise it has the value 0. NW is set to the current value of NEWIT, which is 1 at the beginning of a new nest of iterations, otherwise it is set to 0.

### Location Stack Entry



When a program segment is activated, an entry is pushed to the IS. ORG is set to the starting address of the segment, LEN is set to the length of the segment, DE is set for the DM, and the other three flags are set according to





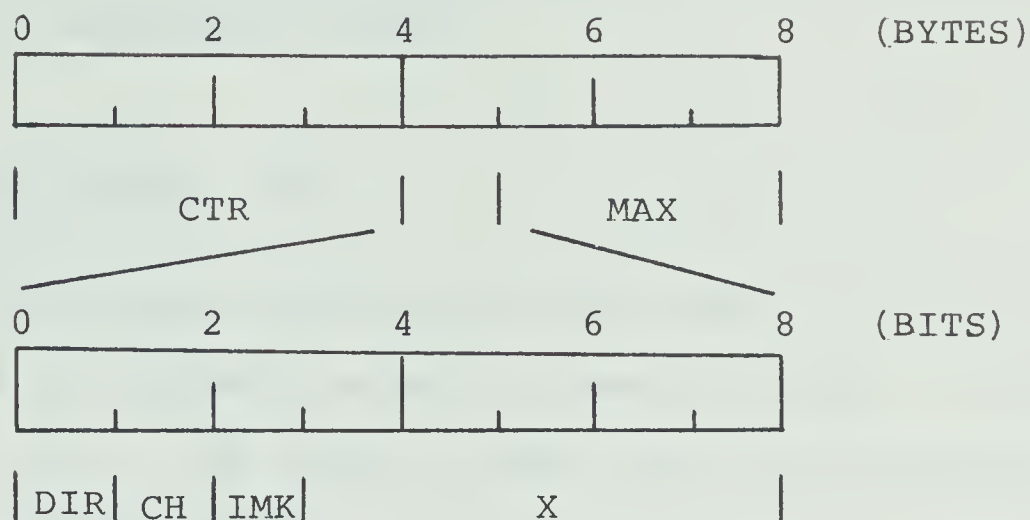
the state of the system at that time. REL is set to 0 to initialize the process of accessing instructions. Each time an instruction is accessed, REL is incremented by the length of that instruction so that the address of the next instruction may be properly calculated by  $ORG + REL$ . When REL has the same value as LEN the segment is terminated by popping the IS. This process automatically re-activates the calling segment exactly as it was when the call was made. The control information in each entry is used to coordinate the IS with the other stacks in the APLM.

#### 3.3.4 Iteration Control Stack. (IS)

IS is used to control the element-by-element evaluation of array-valued expressions, facilitating the array orientation of API programs. Each entry contains a counter (CTR), a direction (DIR), a maximum value (MAX), and control information. CTR is originally set to origin 0 and incremented to the maximum value contained in MAX. The direction of incrementation is supplied by DIR which is 0 for increasing and 1 for decreasing. Control information consists of a change flag (CH), and an iteration mark flag (IMK).



### Iteration Stack Entry



IS behaves like a nest of loops. When a set of iterations is started, MAX and DIR are pushed as an entry to IS, and CTR is initialized. When data is to be operated on, the indexing information is taken from IS and applied to all the instructions in that particular code segment. When a segment is completed, and the required iterations have not been completed, CTR is incremented and the code segment is re-initialized with the new CTR value. When the required iterations have been completed, both LS and IS are popped, returning APIM to its state before the iterations began. For an n-dimensional array, n entries in LS provide the necessary information for accessing the data in the required sequence. IMK functions as a flag to indicate the outermost iteration of a nest of iterations. As the nest of iterations is indexed, CH is set to 1 if the CTR for that



particular IS entry was changed. If CTR was not incremented during that indexing, CH is set to 0. The function of the IS is fully explained in 6.2.

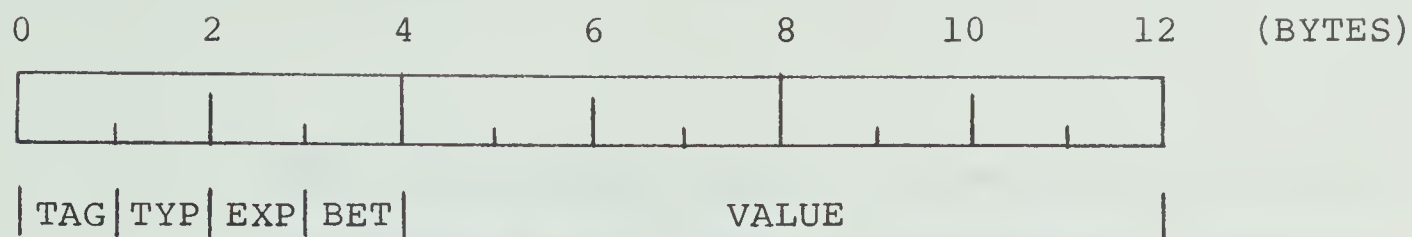
### 3.3.5 Value Stack. (VS)

VS is the main stack in the APLM and is used in the evaluation of expressions and in function calls. Each entry consists of a tag (TAG), a type (TYP), an expression flag (EXP), a beat flag (BET), and a value (VALUE). Permissible tags are JT (j-vector), DT (array descriptor), FDT (function descriptor), FMT (function mark), FT (function descriptor pointer), AT (encoded M-address), NPT (name pointer), RT (reduction accumulator), SGT (segment descriptor), ST (scalar value), UT (unidentified value), NIT (end of ICB block), and NT (for ICB block). Permissible types are BOOL (boolean), CHAR (character), INT (integer), REAL (real), IREAL (long real), and UNUSE (unused memory). VALUE is described by TAG and by TYP the the case of TAG being ST. EXP further describes the QS segment pointed to when TAG is SGT. EXP has the value 2 if the segment is an array-valued expression, 1 if it is an array, and 0 otherwise. BET has the value 0 if the segment is beatable, and 1 otherwise.

The value field of a VS entry serves as an accumulator. Monadic and dyadic operations are carried out on the top and top two entries of VS respectively.



### Value Stack Entry



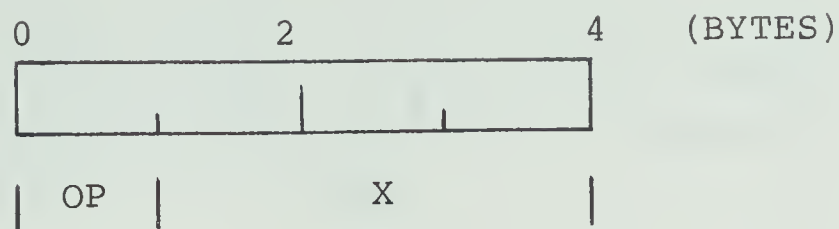
### 3.3.6 Instruction Stack. (QS)

This stack could be more correctly called a buffer because entries other than the latest, or top entry, may be accessed and altered at any time during the operation of both the DM and EM. Instructions to EM are both compiled and altered, if need be, by DM in QS. EM accesses these instructions and interprets them. Instructions in QS consist of entries of lengths one, two, or three words which occur in the following formats.

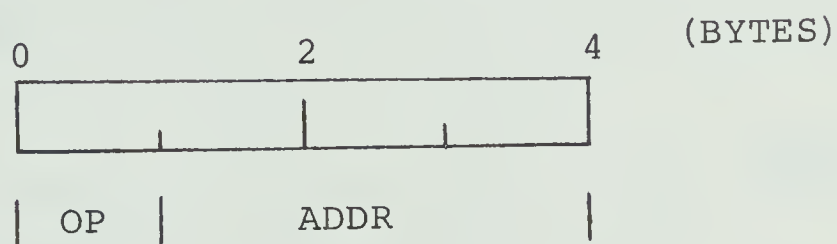
- A. Type 1 consists of a one-word entry which contains only a one byte operation code (OP). Unused fields are labelled X.



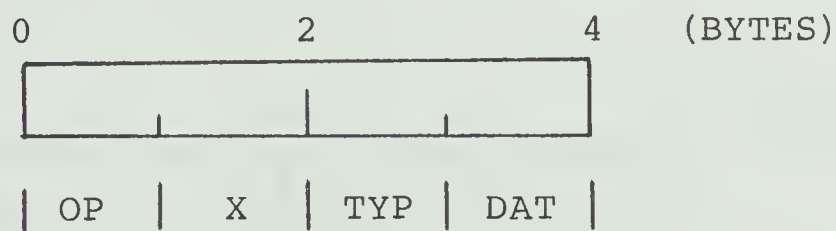




- B. Type 2 consists of a one-word entry which contains an OP and a three-byte address (ADDR).

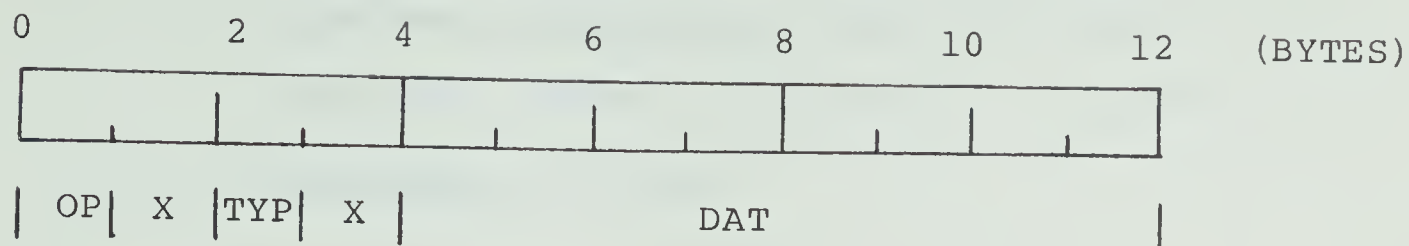


- C. Type 3 contains an CP, a one-byte type-descriptor (TYP), and one bit of logical data or one byte of character data (DAT).

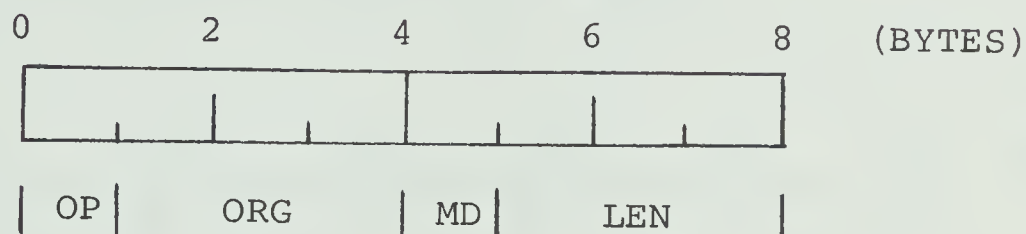


- D. Type 4 is similar to type 3 except that DAT may be a full word integer or real value.

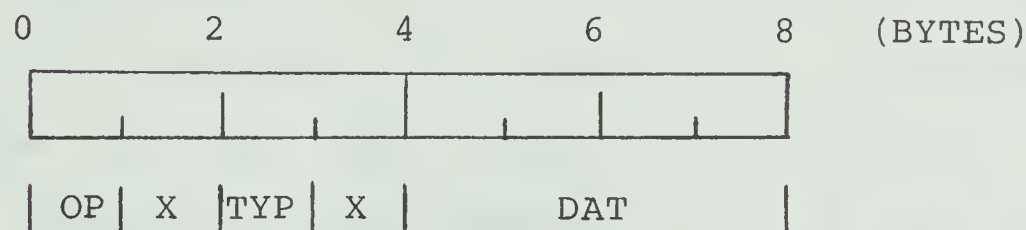




E. Type 5 is two words in length and contains an origin field (CRG), a length field (LEN), a direction field (MD), and an OP. The MD field could contain the direction bit for a j-vector.



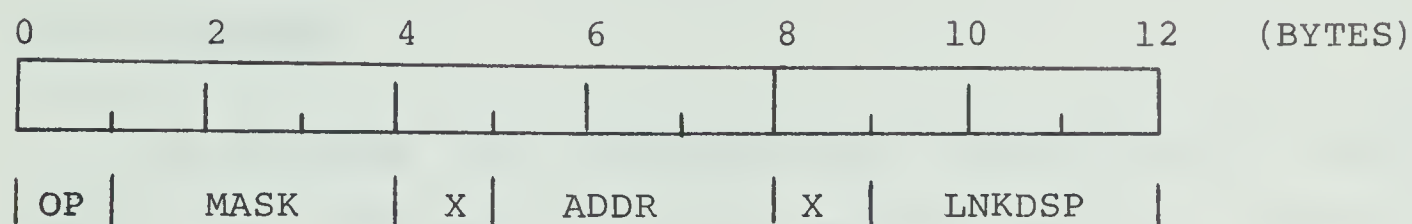
F. Type 6 is three words in length and is similar to type 4 except that IAT may be double precision.



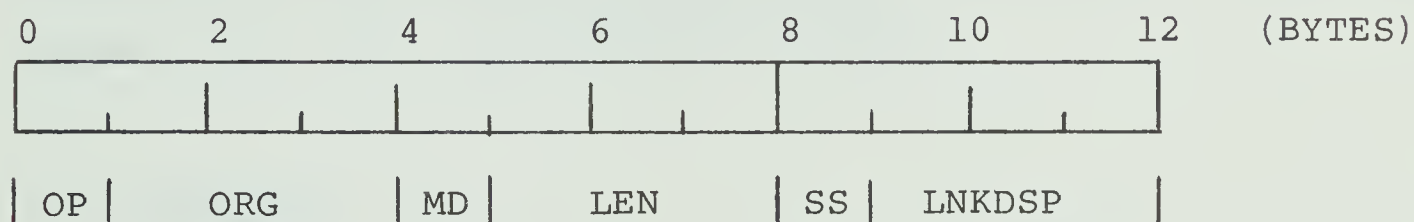
G. Type 7 contains a 24 bit mask (MASK), and a 24-bit link displacement value (LNKDSF). MASK



provides information about the rank of the array and INKDSP is used as a pointer to locations in the QS.



H. Type 8 is the same as type 5 with the addition of a INKDSP field as a suffix.



### 3.3.7 Additional Registers.

The explanation of four single-value registers completes the discussion of register-like structures in APIM. IOFG is the index origin of the currently active function. FBASE is the base address, in M, of the currently active function. FREG is the VS index of the function mark for the currently active function. ISMK is the index of the



topmost IS entry containing 1 in its MARK field.

### 3.4 Data Structures.

#### 3.4.1 Scalars.

The simplest data structures are scalar values. Scalar values in APLM may be of type boolean, character, integer, real, and long real. Scalars are treated as arrays which are of rank 0, but are not stored in memory as arrays. They are stored in VS or as immediate operands in DM source code. Each scalar has a type flag (see 3.3.5), which provides information for encoding and decoding. This type attribute appears with the scalar value in the VS.

#### 3.4.2 Arrays.

The representation of an array consists of two parts. The first part is the actual array values in a dense, row-major, linear allocation in M. The second part of the data structure is an array descriptor (DA). Each array is referenced by at least one DA containing the rank, dimensions, and access-function constants for that array. Each array may be referenced by a different DA for each application of beating to that array, as explained in Chapter Two. DAs are stored in the high-order portion of M.





The DA may now be fully described because the storage access function has been described in Chapter Two. In common with other objects in M, each DA has a standard two-word prefix containing a reference count (RC), a length value (VALUE), and a filler count (FILLER). The first word of the body of the DA contains the base address (VBASE), and the second word contains ABASE. The third word contains the rank (RANK) of the array. A two-word entry for each dimension of the array occurs after the third word. Each two-word entry consists of the dimension (R), and the weighting constant  $DEI(I)$ , labelled D, for that dimension.

### Descriptor Array

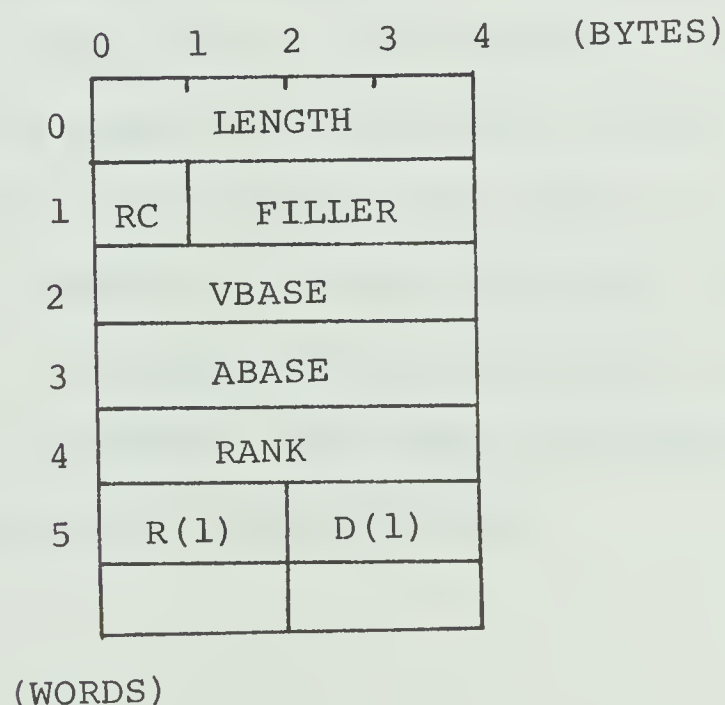


Figure 3.2



### 3.5 Program Segments.

Programs written in APL are compiled by the CM into one or more program segments which contain a series of DM instructions. Segments contain no absolute addresses, and named variables are encoded as INX values which remain constant in the system. Program constants are encoded in program segments and all internal references are relative to the base of the program segment. Programs are therefore relocatable.

Each program segment is referenced by a segment descriptor which has the standard two-word prefix. The first word of the body of the segment descriptor contains the address of the start (FVBASE) of the program segment in M, and the result flag (FRS). The second word contains the length (FIEN), in bytes, and the index origin (FCG) of the segment. The third word contains the number of parameters (FPS), and the number of local variables (FLCIS) in the program segment. A two-byte field containing the INX value for the result, parameters, and local variables, if any, in that order, occurs after the third word.



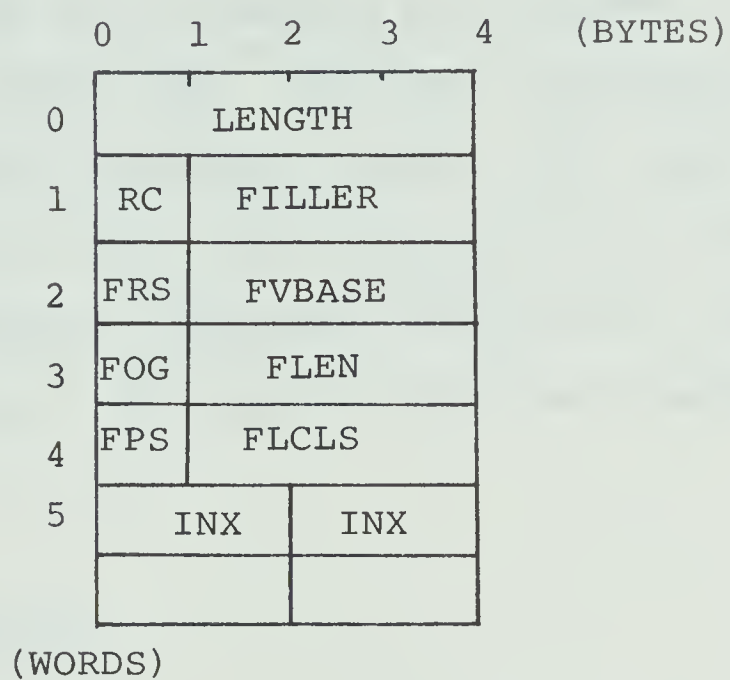
Segment Descriptor.

Figure 3.3



## C-Machine

### Chapter Four

The C-Machine (CM), compiles APL programs into object modules, in a Pclish form, which are interpreted by the D-Machine DM. Program segments contain DM instructions in various formats, which are explained in section 4.4. Section 4.2 contains a condensed overview of the functions of APL operators. Section 4.3 lists and describes all DM instructions.

#### 4.1 CM Object Modules.

A compiled program segment is stored in low-order M and referenced by a segment descriptor, which is stored in high-order M. The segment descriptor is referenced by a entry in NT. All program segments and named objects are accessed by their INX values obtained from NIT. Therefore, all references to named objects or other program segments are replaced by their respective INX values in the CM. Each APL operator is replaced by a DM instruction. Program constants are encoded in the program segments as data arrays. These arrays are constructed in the same format as named arrays of data, and are pointed to by array descriptors. Each program constant array is then replaced by its array descriptor





pointer. Each program segment is relocatable because all internal references are relative to the base of the program segment. Data arrays are also relocatable. An APL program need not be re-compiled each time it is to be executed because the object modules of that program are relocatable and data independent. That is, all data references are pointers to data array descriptors. Arrays may therefore be altered and still remain correctly referenced. Array conformability is checked by the DM when it interprets the CM object modules.

#### 4.2 APL Overview.

It is assumed that the reader has a working knowledge of APL. However, a brief description of the function of APL operators is included in this chapter for reference purposes. Table 4.1 contains the monadic and dyadic primitive scalar functions. Tables 4.2 and 4.3 contain information about inner and outer products respectively. Table 4.4 contains the primitive mixed functions and notes about those functions are listed in table 4.5.



Monadic form $fB$			$f$	Dyadic form $AfB$										
Definition or example	Name			Name	Definition or example									
$+B \leftrightarrow 0+B$	Plus		+	Plus	$2+3.2 \leftrightarrow 5.2$									
$-B \leftrightarrow 0-B$	Negative		-	Minus	$2-3.2 \leftrightarrow -1.2$									
$\times B \leftrightarrow (B>0)-(B<0)$	Signum		$\times$	Times	$2\times 3.2 \leftrightarrow 6.4$									
$\div B \leftrightarrow 1\div B$	Reciprocal		$\div$	Divide	$2\div 3.2 \leftrightarrow 0.625$									
<table><tr><td><math>B</math></td><td><math>\lceil B</math></td><td><math>\lfloor B</math></td></tr><tr><td>3.14</td><td>4</td><td>3</td></tr><tr><td>-3.14</td><td>-3</td><td>-4</td></tr></table>	$B$	$\lceil B$	$\lfloor B$	3.14	4	3	-3.14	-3	-4	Ceiling		$\lceil$	Maximum	$3\lceil 7 \leftrightarrow 7$
$B$	$\lceil B$	$\lfloor B$												
3.14	4	3												
-3.14	-3	-4												
	Floor		$\lfloor$	Minimum	$3\lfloor 7 \leftrightarrow 3$									
$*B \leftrightarrow (2.71828\dots)*B$	Exponential		*	Power	$2*3 \leftrightarrow 8$									
$\odot *N \leftrightarrow N \leftrightarrow *\odot N$	Natural logarithm		$\odot$	Logarithm	$A\odot B \leftrightarrow \text{Log } B \text{ base } A$ $A\odot B \leftrightarrow (\odot B)\div\odot A$									
$ \neg 3.14 \leftrightarrow 3.14$	Magnitude			Residue	<table><tr><td>Case</td><td><math>A B</math></td></tr><tr><td><math>A\neq 0</math></td><td><math>B-( A)\times\lfloor B\div A</math></td></tr><tr><td><math>A=0, B\geq 0</math></td><td><math>B</math></td></tr><tr><td><math>A=0, B&lt;0</math></td><td>Domain error</td></tr></table>	Case	$A B$	$A\neq 0$	$B-( A)\times\lfloor B\div A$	$A=0, B\geq 0$	$B$	$A=0, B<0$	Domain error	
Case	$A B$													
$A\neq 0$	$B-( A)\times\lfloor B\div A$													
$A=0, B\geq 0$	$B$													
$A=0, B<0$	Domain error													
$!0 \leftrightarrow 1$ $!B \leftrightarrow B\times!B-1$ or $!B \leftrightarrow \text{Gamma}(B+1)$	Factorial		!	Binomial coefficient	$A!B \leftrightarrow (!B)\div(!A)\times!B-A$ $2!5 \leftrightarrow 10 \quad 3!5 \leftrightarrow 10$									
$?B \leftrightarrow \text{Random choice from } \neg B$	Roll		?	Deal	A Mixed Function (See Table 3.8)									
$\circ B \leftrightarrow B\times 3.14159\dots$	Pi times		$\circ$	Circular	See Table at left									
$\sim 1 \leftrightarrow 0 \quad \sim 0 \leftrightarrow 1$	Not		$\sim$											

<table><tr><td><math>(-A)\circ B</math></td><td><math>A</math></td><td><math>A\circ B</math></td></tr><tr><td><math>(1-B*2)*.5</math></td><td>0</td><td><math>(1-B*2)*.5</math></td></tr><tr><td><math>\text{Arcsin } B</math></td><td>1</td><td><math>\text{Sine } B</math></td></tr><tr><td><math>\text{Arccos } B</math></td><td>2</td><td><math>\text{Cosine } B</math></td></tr><tr><td><math>\text{Arctan } B</math></td><td>3</td><td><math>\text{Tangent } B</math></td></tr><tr><td><math>(-1+B*2)*.5</math></td><td>4</td><td><math>(1+B*2)*.5</math></td></tr><tr><td><math>\text{Arcsinh } B</math></td><td>5</td><td><math>\text{Sinh } B</math></td></tr><tr><td><math>\text{Arccosh } B</math></td><td>6</td><td><math>\text{Cosh } B</math></td></tr><tr><td><math>\text{Arctanh } B</math></td><td>7</td><td><math>\text{Tanh } B</math></td></tr></table>	$(-A)\circ B$	$A$	$A\circ B$	$(1-B*2)*.5$	0	$(1-B*2)*.5$	$\text{Arcsin } B$	1	$\text{Sine } B$	$\text{Arccos } B$	2	$\text{Cosine } B$	$\text{Arctan } B$	3	$\text{Tangent } B$	$(-1+B*2)*.5$	4	$(1+B*2)*.5$	$\text{Arcsinh } B$	5	$\text{Sinh } B$	$\text{Arccosh } B$	6	$\text{Cosh } B$	$\text{Arctanh } B$	7	$\text{Tanh } B$	<table><tr><td><math>\wedge</math></td><td>And</td><td><math>A B</math></td><td><math>A\wedge B</math></td><td><math>A\vee B</math></td><td><math>A\nabla B</math></td><td><math>A\nabla B</math></td></tr><tr><td><math>\vee</math></td><td>Or</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td><math>\nabla</math></td><td>Nand</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td><math>\nabla</math></td><td>Nor</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td></td><td></td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	$\wedge$	And	$A B$	$A\wedge B$	$A\vee B$	$A\nabla B$	$A\nabla B$	$\vee$	Or	0	0	0	1	1	$\nabla$	Nand	0	1	0	1	0	$\nabla$	Nor	1	0	0	1	0			1	1	1	0	0	<table><tr><td><math>&lt;</math></td><td>Less</td><td rowspan="6">Relations Result is 1 if the relation holds, 0 if it does not: <math>3\leq 7 \leftrightarrow 1</math> <math>7\leq 3 \leftrightarrow 0</math></td></tr><tr><td><math>\leq</math></td><td>Not greater</td></tr><tr><td><math>=</math></td><td>Equal</td></tr><tr><td><math>\geq</math></td><td>Not less</td></tr><tr><td><math>&gt;</math></td><td>Greater</td></tr><tr><td><math>\neq</math></td><td>Not Equal</td></tr></table>	$<$	Less	Relations Result is 1 if the relation holds, 0 if it does not: $3\leq 7 \leftrightarrow 1$ $7\leq 3 \leftrightarrow 0$	$\leq$	Not greater	$=$	Equal	$\geq$	Not less	$>$	Greater	$\neq$	Not Equal
$(-A)\circ B$	$A$	$A\circ B$																																																																											
$(1-B*2)*.5$	0	$(1-B*2)*.5$																																																																											
$\text{Arcsin } B$	1	$\text{Sine } B$																																																																											
$\text{Arccos } B$	2	$\text{Cosine } B$																																																																											
$\text{Arctan } B$	3	$\text{Tangent } B$																																																																											
$(-1+B*2)*.5$	4	$(1+B*2)*.5$																																																																											
$\text{Arcsinh } B$	5	$\text{Sinh } B$																																																																											
$\text{Arccosh } B$	6	$\text{Cosh } B$																																																																											
$\text{Arctanh } B$	7	$\text{Tanh } B$																																																																											
$\wedge$	And	$A B$	$A\wedge B$	$A\vee B$	$A\nabla B$	$A\nabla B$																																																																							
$\vee$	Or	0	0	0	1	1																																																																							
$\nabla$	Nand	0	1	0	1	0																																																																							
$\nabla$	Nor	1	0	0	1	0																																																																							
		1	1	1	0	0																																																																							
$<$	Less	Relations Result is 1 if the relation holds, 0 if it does not: $3\leq 7 \leftrightarrow 1$ $7\leq 3 \leftrightarrow 0$																																																																											
$\leq$	Not greater																																																																												
$=$	Equal																																																																												
$\geq$	Not less																																																																												
$>$	Greater																																																																												
$\neq$	Not Equal																																																																												

Table of Dyadic  $\circ$  Functions

APL Scalar Functions

Figure 4.1

Reprinted by permission from APL\360; User's Manual

@1968 by International Business Machines Corporation.



$\rho A$	$\rho B$	$\rho Af.gB$	Conformability requirements	Definition $Z \leftarrow Af.gB$
	$V$			$Z \leftarrow f/AqB$
$U$	$V$			$Z \leftarrow f/AgB$
$U$	$V$			$Z \leftarrow f/AgB$
	$V$	$W$	$U=V$	$Z \leftarrow f/AqB$
	$V$	$W$		$Z[I] \leftarrow f/AgB[;I]$
$T$	$U$	$T$		$Z[I] \leftarrow f/A[I;]gB$
	$V$	$W$	$U=V$	$Z[I] \leftarrow f/AgB[;I]$
$T$	$U$	$T$	$U=V$	$Z[I] \leftarrow f/A[I;]gB$
$T$	$U$	$T$	$U=V$	$Z[I;J] \leftarrow f/A[I;]gB[;J]$

## APL Inner Products

Figure 4.2

$\rho A$	$\rho B$	$\rho A \circ .gB$	Definition $Z \leftarrow A \circ .gB$
	$V$	$V$	$Z \leftarrow AgB$
$U$	$V$	$U$	$Z[I] \leftarrow AgB[I]$
$U$	$V$	$U$	$Z[I] \leftarrow A[I]gB$
	$V$	$U$	$Z[I;J] \leftarrow A[I]gB[J]$
	$V$	$W$	$Z[I;J] \leftarrow AgB[I;J]$
$T$	$U$	$T$	$Z[I;J] \leftarrow A[I;J]gB$
$U$	$V$	$W$	$Z[I;J;K] \leftarrow A[I]gB[J;K]$
$T$	$U$	$T$	$Z[I;J;K] \leftarrow A[I;J]gB[K]$
$T$	$U$	$T$	$Z[I;J;K;L] \leftarrow A[I;J]gB[K;L]$

## APL Outer Products

Figure 4.3

Reprinted by permission from APL\360; User's Manual

@1968 by International Business Machines Corporation.



Name	Sign <sup>1</sup>	Definition or example <sup>2</sup>
Size	$\rho A$	$\rho P \leftrightarrow 4$ $\rho E \leftrightarrow 3 \ 4$ $\rho 5 \leftrightarrow 10$
Reshape	$V \rho A$	Reshape $A$ to dimension $V$ $3 \ 4 \rho 12 \leftrightarrow E$ $12 \rho E \leftrightarrow 112$ $0 \rho E \leftrightarrow 10$
Ravel	$,A$	$,A \leftrightarrow (\times/\rho A) \rho A$ $,E \leftrightarrow 112$ $\rho, 5 \leftrightarrow 1$
Catenate	$V, V$	$P, 12 \leftrightarrow 2 \ 3 \ 5 \ 7 \ 1 \ 2$ $'T', 'HIS' \leftrightarrow 'THIS'$
Index <sup>34</sup>	$V[A]$ $M[A;A]$ $A[A;..]$ $..;A]$	$P[2] \leftrightarrow 3$ $P[4 \ 3 \ 2 \ 1] \leftrightarrow 7 \ 5 \ 3 \ 2$ $E[1 \ 3; 3 \ 2 \ 1] \leftrightarrow 3 \ 2 \ 1$ $11 \ 10 \ 9$ $E[1;] \leftrightarrow 1 \ 2 \ 3 \ 4$ $ABCD$ $E[;1] \leftrightarrow 1 \ 5 \ 9$ $'ABCDEFGHijkl'[E] \leftrightarrow EFGH$ $ijkl$
Index generator <sup>3</sup>	$1S$	First $S$ integers $14 \leftrightarrow 1 \ 2 \ 3 \ 4$ $10 \leftrightarrow$ an empty vector
Index of <sup>3</sup>	$V1A$	Least index of $A$ in $V$ , or $1+\rho V$ $P13 \leftrightarrow 2$ $5 \ 1 \ 2 \ 5$ $P1E \leftrightarrow 3 \ 5 \ 4 \ 5$ $4 \ 414 \leftrightarrow 1$ $5 \ 5 \ 5 \ 5$
Take	$V \uparrow A$	Take or drop $ V[I] $ first elements of coordinate $I$ $2 \ 3 \uparrow X \leftrightarrow ABC$ $(V[I] \geq 0)$ or last $(V[I] < 0)$ elements of coordinate $I$ $EFG$
Drop	$V \downarrow A$	$-2 \uparrow P \leftrightarrow 5 \ 7$
Grade up <sup>35</sup>	$\Delta A$	The permutation which would order $A$ (ascending or descending) $\Delta 3 \ 5 \ 3 \ 2 \leftrightarrow 4 \ 1 \ 3 \ 2$
Grade down <sup>35</sup>	$\nabla A$	$\nabla 3 \ 5 \ 3 \ 2 \leftrightarrow 2 \ 1 \ 3 \ 4$
Compress <sup>5</sup>	$V/A$	$1 \ 0 \ 1 \ 0/P \leftrightarrow 2 \ 5$ $1 \ 0 \ 1 \ 0/E \leftrightarrow 5 \ 7$ $9 \ 11$ $1 \ 0 \ 1/[1]E \leftrightarrow 1 \ 2 \ 3 \ 4 \leftrightarrow 1 \ 0 \ 1/E$ $9 \ 10 \ 11 \ 12$
Expand <sup>5</sup>	$V \setminus A$	$1 \ 0 \ 1 \setminus 12 \leftrightarrow 1 \ 0 \ 2$ $1 \ 0 \ 1 \ 1 \ 1 \setminus X \leftrightarrow A \ BCD$ $E \ FGH$ $I \ JKL$
Reverse <sup>5</sup>	$\phi A$	$\phi X \leftrightarrow DCBA$ $\phi[1]X \leftrightarrow \ominus X \leftrightarrow EFGH$ $HGFE$ $\phi P \leftrightarrow 7 \ 5 \ 3 \ 2$ $ABCD$ $LKJI$
Rotate <sup>5</sup>	$A \phi A$	$3 \phi P \leftrightarrow 7 \ 2 \ 3 \ 5 \leftrightarrow -1 \phi P$ $1 \ 0 \ -1 \phi X \leftrightarrow BCDA$ $EFGH$ $LIJK$
Transpose	$V \oslash A$ $\oslash A$	Coordinate $I$ of $A$ becomes coordinate $V[I]$ of result $2 \ 1 \oslash X \leftrightarrow AEI$ $BFJ$ $CGK$ $DHL$ $1 \ 1 \oslash E \leftrightarrow 1 \ 6 \ 11$ $\oslash E \leftrightarrow 2 \ 1 \oslash E$
Membership	$A \in A$	$\rho W \in Y \leftrightarrow \rho W$ $E \in P \leftrightarrow 0 \ 1 \ 1 \ 0$ $P \in 14 \leftrightarrow 1 \ 1 \ 0 \ 0$ $1 \ 0 \ 1 \ 0$ $0 \ 0 \ 0 \ 0$
Decode	$V1V$	$1011 \ 7 \ 7 \ 6 \leftrightarrow 1776$ $24 \ 60 \ 6011 \ 2 \ 3 \leftrightarrow 3723$
Encode	$V \uparrow S$	$24 \ 60 \ 60 \uparrow 3723 \leftrightarrow 1 \ 2 \ 3$ $60 \ 60 \uparrow 3723 \leftrightarrow 2 \ 3$
Deal <sup>3</sup>	$S?S$	$W?Y \leftrightarrow$ Random deal of $W$ elements from $1Y$

## APL Mixed Functions

Figure 4.4







1. Restrictions on argument ranks are indicated by: *S* for scalar, *V* for vector, *M* for matrix, *A* for Any. Except as the first argument of *S*<sub>1</sub>*A* or *S*[*A*], a scalar may be used instead of a vector. A one-element array may replace any scalar.
2. Arrays used  
in examples:     *P* ↔ 2 3 5 7     *E* ↔ 

1	2	3	4
5	6	7	8
9	10	11	12

*X* ↔ 

<i>ABCD</i>
<i>EFGH</i>
<i>IJKL</i>
3. Function depends on index origin.
4. Elision of any index selects all along that coordinate.
5. The function is applied along the last coordinate; the symbols */*, *\*, and *⊖* are equivalent to */*, *\*, and *⊕*, respectively, except that the function is applied along the first coordinate. If [*S*] appears after any of the symbols, the relevant coordinate is determined by the scalar *S*.

#### Notes To Figure 4.4

#### Figure 4.5

Reprinted by permission from APL\360; User's Manual  
 ©1968 by International Business Machines Corporation.



### 4.3 Object Module Instructions.

Instructions for the DM fall into eight separate classes and appear as a series of instructions in a stack. Following is a discussion of the instructions and their description, for each class.

#### 4.3.1 Storage Management Instructions.

These instructions load scalar values, segment descriptor pointers, and j-vectors into the value stack (VS). They produce no instructions for the EM nor do they result in any mathematical or logical operations on data.

<u>Opcode</u>	<u>Operand</u>	<u>Description</u>
LDS	scalar	The scalar value in the operand field is to be loaded into a new entry in the VS with their appropriate type value.
LDSEG	seg-desc	The segment descriptor address in the operand field is to be loaded as a new entry in the VS.
LDJ	j-code	The j-vector in the operand field is to be loaded as a new



		entry in the VS.
IDIS	K	The entry K from the top of the IS is to be accessed and its counter value is to be loaded as a new entry in the VS.
LDCCN	K	Access the program constant which is located at FBASE+K. An array descriptor is constructed to access that constant (element or array) and its address is pushed to the VS with tag FDI.
LDN	N	The address of the object N is pushed to the VS.
LDNF	N	The variable object N is referred to and its current value is transferred from NT to VS.

The following instructions produce EM instructions which assign values to the corresponding variables.

<u>Opcode</u>	<u>Operand</u>	<u>Description</u>
ASGN	←	The value in the second entry



of the VS is stored in the variable whose address is contained in the first entry of VS. The value is then discarded by popping VS.

ASGNV                      ←      A procedure similar to that for ASGN is followed except that the value is left in VS.

#### 4.3.2 Control Instructions.

These instructions cause the DM to continue accessing instructions at an updated address.

<u>Opcode</u>	<u>Operand</u>	<u>Description</u>
JMP	K	Continue with normal instruction accessing at the location K (signed) bytes from the present location.
JMP0	K	Check last entry in the VS. If the value is 0, do the same as JMP K. Otherwise, continue with normal instruction addressing.
JMP1	K	Check last entry in VS. If the value is 1, do the same as





	JMF K, otherwise continue with normal accessing of instructions.
LEAVE	Terminate interpretation of the current program segment.
RETURN	Terminate interpretation of the current program segment and resume interpreting the calling program segment.
DO	Call the EM to interpret instructions in the QS.
DOI	Call the EM and allocate temporary space for a result, if any, and leave the result on the VS.

#### 4.3.3 Dyadic Scalar Operators.

These DM instructions implement the dyadic scalar operators of AFL. Dyadic operators require that both operands be the same type and dimension or that one operand be a scalar of the same type as the other operand. In the second case, the scalar value is in effect extended to the same dimension as the other operand.



<u>DM Operator</u>	<u>API Symbol</u>	<u>Definition</u>
ADD	+	Add.
SUB	-	Subtract.
MUL	×	Multiply.
DIV	÷	Divide.
MCD		Residue.
MIN	L	Minimum.
MAX	┌	Maximum.
PWR	*	Power.
LOG	⊗	Logarithm.
CIR	○	Circular function.
DEAL	?	Random deal.
COMB	!	Binomial coefficient or beta function.
AND	^	Logical and.
OR	∨	Logical or.
NAND	⋈	Logical nand.
NOR	⋈	Logical nor.
LT	<	Less than.
LE	≤	Less than or equal.
EQ	=	Equal.
GE	≥	Greater than or equal.
GT	>	Greater than.
NE	≠	Not equal.



#### 4.3.4 Monadic Scalar Operators.

These instructions implement the same functions as monadic scalar APL operators. Each operator is applied to one operand, which may be a scalar value or an array.

<u>DM Operator</u>	<u>APL Symbol</u>	<u>Definition</u>
PLUS	+	Plus.
MINUS	-	Minus.
SGN	×	Signum.
RECIP	÷	Reciprocal.
ABS		Absolute value.
FLOOR	⌊	Floor.
CEIL	⌈	Ceiling.
EX	*	Exponential (base e).
LCGE	⊗	Logarithm (base e).
PI	○	Pi times.
RAND	?	Random deal.
FAC	!	Factorial or gamma function.
NOT	~	Logical not.



#### 4.3.5 Selection Operators.

These instructions result in arrays being re-shaped or re-arranged without individual elements being changed.

<u>DM Operator</u>	<u>API Symbol</u>	<u>Definition</u>
TAKE	$\uparrow$	Take.
DROP	$\downarrow$	Drop.
REV K	$\phi$	Reverse along coordinate K.
TRANS	$\Phi$	Generalized transpose.
INX K	[	Index on coordinate K.

#### 4.3.6 Immediate Evaluation Operators.

These operations must be executed as soon as they are encountered by the DM, because it may be impossible or difficult to apply the processes of drag-along and heating.

<u>DM Operator</u>	<u>API Symbol</u>	<u>Definition</u>
BASE	$\perp$	Base value (decode).
REP	$\tau$	Representation (encode).
GDU	$\Delta$	Grade up.
GDD	$\nabla$	Grade down.
CAT K	,	Catenate K entries.
RAV	,	Ravel.





URHO	$\rho$	Dimension.
DRHO	$\rho$	restructure.
UIOTA	$\iota$	Interval.

#### 4.3.7 Deferrable Operators.

The execution of these instructions may be deferred in the instruction stack (QS).

<u>DM operator</u>	<u>APL Symbol</u>	<u>Definition</u>
ROT K	$\phi$	Rotate on coordinate K.
EPS	$\epsilon$	Membership.
DICTA	$\iota$	Rank.
CMPRS K	/	Compress on coordinate K.
EXPND	\	Expand on coordinate K.
SUBS K	[	Subscript with K entries in VS.

#### 4.3.8 Compound Operators.

These instructions are compound APL operations, where AOP means an APL operator instruction, and OF is the operand.







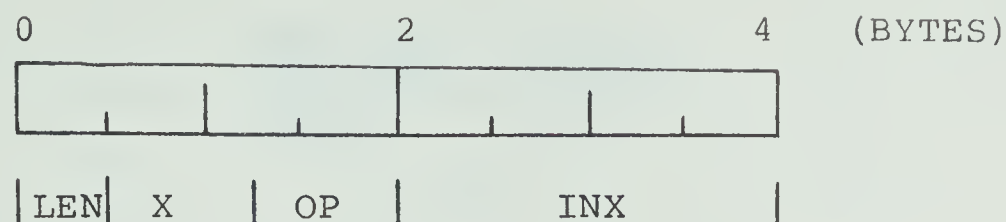










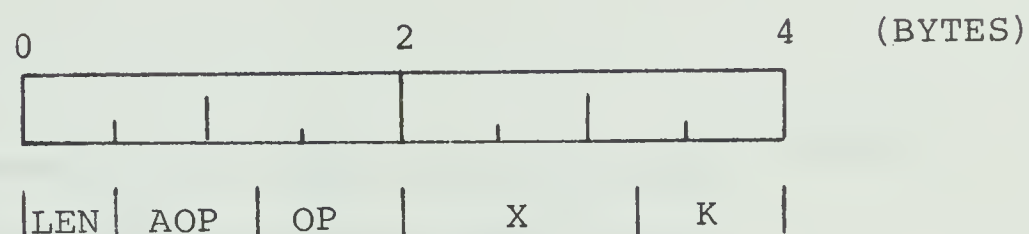


CPEFATOR      CPCCDE

IDN              24

IDNF             25

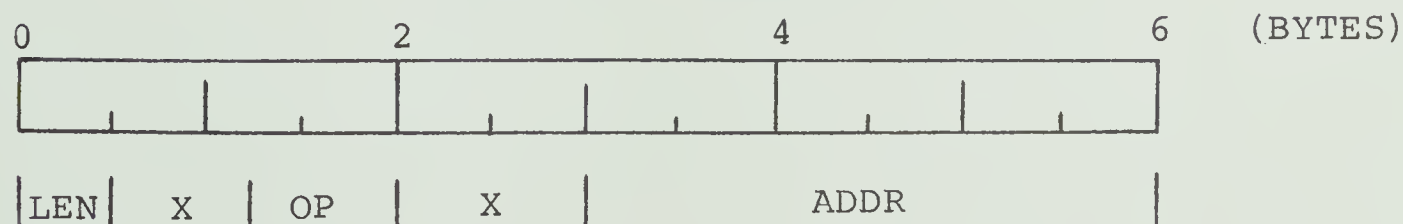
E. Type 5 is a full word in length and contains an ACP and a K.



OPEFATOR      CPCCDE

RED K OF        26

F. Type 6 is six bytes in length and contains an address field (ADDR).

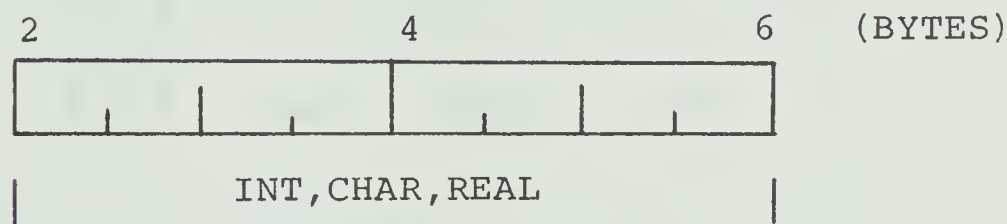
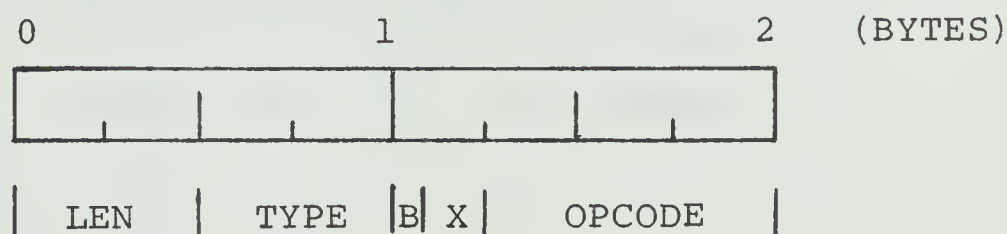




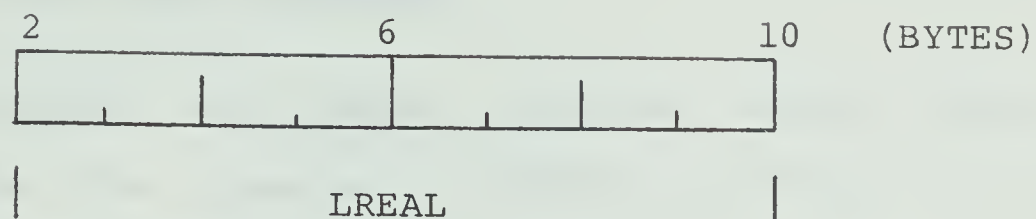
OPERATOR	CFCODE
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17
18	18
19	19
20	20
21	21
22	22
23	23
24	24
25	25
26	26
27	27
28	28
29	29
30	30
31	31
32	32
33	33
34	34
35	35
36	36
37	37
38	38
39	39
40	40
41	41
42	42
43	43
44	44
45	45
46	46
47	47
48	48
49	49
50	50
51	51
52	52
53	53
54	54
55	55
56	56
57	57
58	58
59	59
60	60
61	61
62	62
63	63
64	64
65	65
66	66
67	67
68	68
69	69
70	70
71	71
72	72
73	73
74	74
75	75
76	76
77	77
78	78
79	79
80	80
81	81
82	82
83	83
84	84
85	85
86	86
87	87
88	88
89	89
90	90
91	91
92	92
93	93
94	94
95	95
96	96
97	97
98	98
99	99
100	100

LDSEG 27

G. Type 7 varies from two bytes to ten bytes in length and contains a type field (TYP) and any one of the following data fields; binary (B), integer (INT), character (CHAR), real (REAL), and long real (IREAL). The instruction lengths for E, CHAR, INT, REAL, and IREAL are 2, 2, 6, 6, and 10 bytes respectively as shown in the following multiple diagram.

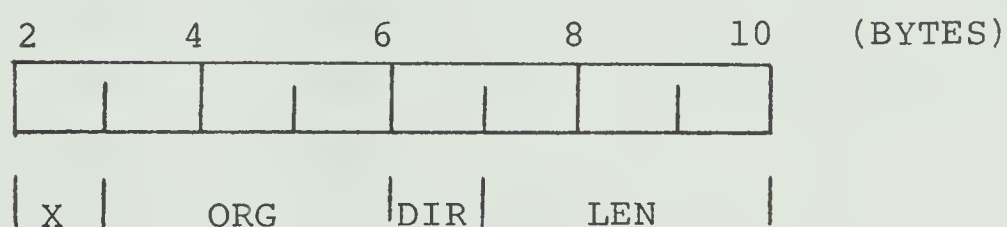
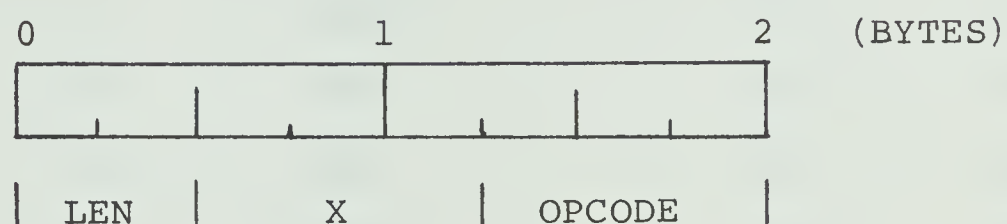






<u>CFFATOR</u>	<u>CPCCDE</u>
IDS	28

- H. Type 8 is ten bytes in length and contains j-vectors in its value field which are encoded as an origin (ORG), a direction (DIF), and a length (JLEN).



<u>CFFATOR</u>	<u>CPCCDE</u>
IDS	29



#### 4.5 Monadic and Dyadic Opcodes.

The monadic and dyadic operators which appear as operands for the operators MCNADIC and DYADIC are listed below with their corresponding opcodes.

<u>DYADIC</u>	<u>CFCODE</u>	<u>MONADIC</u>	<u>DYADIC</u>	<u>CFCODE</u>
ADD	1	PLUS	AND	E
SUB	2	MINUS	OR	F
MUL	3	SIGNUM	NAND	10
DIV	4	RECIP	NOR	11
MCD	5	ABS	LT	12
MIN	6	FLOOR	LE	13
MAX	7	CEIL	EQ	14
PWR	8	EXP	GE	15
LOG	9	LOGE	GT	16
CIR	A	PI	NE	17
DEAL	B	RAND		
CCMB	C	FAC		
	D	NOT		





## The D-Machine

### Chapter Five

#### 5.1 Introduction

The D-Machine (DM), accesses instructions from CM object modules (program segments), interprets them and compiles a series of EM instructions in the instruction buffer (QS). DM uses the location stack (LS) to access DM instructions and program segments. The value stack (VS), provides the necessary information for allocating and accessing memory. The name table (NT) provides the DM with the link between the index value and the descriptor of a named object in memory. Information regarding dimensions of arrays is placed on the iteration stack (IS). The DM applies the processes of drag-along and beating to EM instructions residing in the QS. Some array conformability checking is undertaken by the DM. DM decides when to pass control to the EM.

#### 5.2 DM Object Instructions.

Because the target of the DM is EM instructions it is necessary to be familiar with the EM instruction set, which is explained in the following three sub-sections.



### 5.2.1 Simple Instructions.

<u>Opcode</u>	<u>Name and Function</u>
S	Load scalar: The scalar value is pushed to VS with the tag set to ST.
IFA	Load array element: The index environment for the array is linked to the instruction in the QS. The instruction is changed to FA.
FA	Load array element: An element determined by the indexing environment is pushed to the VS with the tag ST.
IA	Load array address: The index environment for the array is linked to the instruction in the QS, which is then changed to A.
A	Load array address: The encoded address of the selected element is pushed to VS with the tag AT.
IJ	Load j-vector element: The index environment for the j-vector is linked to the instruction in the QS, which is then changed to J.
J	Load j-vector element: The required



element of the j-vector is computed and pushed to VS with the tag ST.

**GOP**            Scalar operator: The value field contains the name of a dyadic scalar arithmetic operator whose operands are the top entries of VS. The result is left in VS after the operands are deleted.

**OP**            Scalar operator: The value field contains the name of a dyadic scalar arithmetic operator whose operand is the top entry of VS. The result is left in VS after the operand is deleted.

**NIL**           Nil: No operation is performed.

**IRD**           Result dimension: This instruction is used only by the DM and is left in QS when a segment is passed to the EM. When encountered by the EM, an IFD is changed to NIL.

**IRP**           Result dimension: This instruction causes the same action as IRD.

### 5.2.2 Control Instructions.

These instructions alter the normal instruction



accessing procedures as well as activate program segments.

<u>Opcode</u>	<u>Name and Function</u>
SGV	Load segment descriptor: The value field contains a QS segment descriptor. This address is made absolute and pushed to VS with the tag SGT.
SG	Activate segment: The value field, if non-zero, points to a pseudo-instruction stack in QS. EM is then activated to execute the stack of instructions.
JMP	Jump: Unconditional jump to the location defined by the link field and the value field.
JN0	Jump if 0: Jump if top of VS contains the value 0.
JN1	Jump if 1: Jump if top of VS contains the value 1.
J0	Jump if 0: Jump if top of VS contains the value 0 and pop the VS.
J1	Jump if 1: Jump if top of VS contains the value 1 and pop the VS.
RED	Begin reduction: Push an element with tag RT to the VS to act as a reduction accumulator and jump as described by the





instruction JMP.

MIT

Mark and iterate: Scalar values on top of VS are used to start a new iteration nest in IS. The absolute value of the VS value, less 1, is the MAX field in IS, the iteration direction (DIR) is forward (0) if VS is positive otherwise backwards (1). The CNT field is initialized to 0 or MAX if DIR is 0 or 1 respectively. The first entry in IS has its MRK bit set to 1 while all other entries are set to 0. Each VS entry is popped. When an SGT entry is found, it is popped and the named segment is activated in IS.

### 5.2.3 Micro-instructions.

These instructions are used primarily for manipulating pseudo-iteration stacks.

<u>Opcode</u>	<u>Name and Function</u>
POP	Pop: The top entry in the VS is popped.
DUP	Duplicate: The link elements of the top VS entry are pushed as a new entry to the VS.



ORG            load Iorg: The current value of ICRG is pushed to the VS with the tag ST.

CY            Cycle: The IS is stepped and the current segment is re-executed if the IS has not overflowed.

LVE           Leave: The current segment is deactivated. Any associated IS entries are erased.

RPT           Repeat: The current segment is re-executed from the beginning without affecting the IS.

CAS           Case: The top entry of VS is expected to contain an integer scalar, say N. The VS is popped and the instruction at the location referenced by N is executed. Execution is then resumed at the location referenced by the VALUE and LINK fields.

VXC           Exchange: The top two entries on the VS are interchanged.

LX1           Load from pseudo-IS: LINK fields are relative pointers to XT entries. The X1 field of the referenced entry is pushed to the VS with the tag ST. The X1 field corresponds to the CTR field of an IS



entry.

LX2            Load from pseudo-IS: The X2 field of the referenced entry is pushed to the VS with the tag ST. The X2 field corresponds to the MAX field of an IS entry.

SX1            Store in pseudo-IS: The top (ST) entry on the VS is stored in the X1 field of the referenced XT entry and the VS is popped.

SX2            Store pseudo-IS: The top (ST) entry of the VS is stored in the X2 field of the referenced XT entry and the VS is popped.

IXL            Index load: IXL initializes the XL instruction by setting the LINK field to point to the IS or a pseudo-IS element.

XL             Index load: This instruction gets the current iteration value, adds IORG and pushes the result to VS with the tag ST.

XS             Index store: IORG is subtracted from the ST entry on the VS and stored in the X1 field of the XT entry at the location described by the VALUE field and the LINK field.



XC                    Index change: The X3 field of the referenced XT entry is set to 1. The X3 field corresponds to the CH field of an IS entry.

ISC                   Activate segment conditional: This instruction initializes the SC instruction.

SC                    Activate segment conditional: The value field of the instruction is a QS segment descriptor. If the bit in the referenced IS or pseudo-IS is 1, the segment is activated. Otherwise, the change bit of the XT entry referenced by the following instruction is set to 0 and this instruction is skipped.

### 5.3 Interpretation of DM-instructions.

The DM interprets DM instructions, which are located in program segments, and compiles EM instructions for all instructions except storage management or control instructions. The following sub-sections of this section contain descriptions of the interpretation of the DM instructions.





### 5.3.1 Storage Management Instructions.

These instructions are primarily concerned with the allocation of storage, and the accessing of data. The instructions IDS, IDSEG, LDJ, and LDN have immediate operands which are pushed to the VS with the tags ST, SGT, JT, and NPT respectively. The LDIS instruction, with operand K, accesses the IS entry K entries from the top and loads in the VS, as a scalar, the value of the CNT field. TAG is set to ST, for a scalar, and TYPE is set to INT, for an integer. The LDNF instruction refers to a named object whose index value is the operand. NT is searched associatively for that INX value and the current value from NT is pushed to the VS. If the NT entry has the tag DT (an array), the reference count of the DA is increased by 1 when it is pushed to the VS, and the TAG is set to FDT. The LDCCN instruction refers to a constant array which is encoded in the corresponding program segment. The operand K is the displacement of the DA from the base of the program segment. The DA is then copied to M with its VEASE set to the base of the function (FBASE), and its ABASE set to the operand K. The DA pointer is then pushed to VS with the tag FDT.

ASGN and ASGNV are operators corresponding to the left arrow in API, but are included as storage management



instructions because they cause values to be stored. The top of the VS is expected to contain a name with tag NPT, or a QS descriptor pointing to a segment containing only an IA instruction. The second entry in the VS points to the segment of code for the right-hand side of the assignment. Several actions are taken by the DM depending on the contents of the VS.

The immediate assignment, that is, storage of the scalar value is NT, or storage of the array element in the array in M, is done under the following situations.

- A. The tag of the top VS entry is NPT and the tag of the second top VS entry is ST.
- B. The tag of the top VS entry is SGT, the deferred expression has one element, and the tag of the second top VS entry is ST.
- C. The tag of the top VS entry is NPT, the tag of the second top VS entry is SGT, and the deferred segment is either a j-vector or a single DA with a reference count of 1 and a value whose reference count is 1.

If none of the previous conditions are identified, then an array is to be assigned. If the tag of the top VS entry is NPT then space is allocated for a DA of the size



necessary to store the result. The assignment is then deferred in the QS as for scalar arithmetic operators.

The only case left occurs when the tag in the top VS entry is SGT. The ranks and the dimensions of the operands are checked for conformability. If the left-hand operand is a j-vector, it must be explicitly evaluated. If the right-hand side contains occurrences of various permutations of the left-hand operand, then the right-hand side is evaluated to temporary space. Finally, the assignment is deferred in the QS.

### 5.3.2 Control Instructions.

The DM control instructions alter the normal instruction accessing sequence. Interpretation of JMP results in changing the REL field of the IS to the updated instruction address. JMP0 and JMP1 are interpreted the same way only if the top VS entry has the value 0 or 1 respectively. LEAVE pops the IS, and the IS if the segment is involved in an iteration. This in effect terminates execution of that particular segment. RETURN is interpreted similarly except that the local variables for the segment are erased from NT.

Interpretation of DC depends on the value in the top entry of the VS. If that value is a scalar, then DC acts as





a no-operator. If the tag is SGT, then the segment, pointed to by the VALUE field of the top entry of the VS, is activated by pushing the segment descriptor to the IS. The top VS entry is then popped. If the tag is NFI, the corresponding NT tag is examined. If it is FT, then the named function is activated. All other cases act as no-operator.

A named function is activated in the following manner:

- A. The segment descriptor for that segment is fetched and is expected to contain the INX values for the result, the parameters, and the local variables. It is also expected that all parameters to the segment have been evaluated and placed in VS with the topmost entry being the leftmost parameter in the APL program function header. These conditions are verified and an error is signaled if any condition is not fulfilled. If no error exists, the DM builds an NT entry for each local variable to the function. Each new tag in NT is set to UT (undefined), unless it corresponds to a parameter. As each INX value is placed in its NT entry, the associated entry is popped from the VS.
- B. A function mark (tag FMT) entry is pushed to VS. The entry contains an encoding of the current values of





FREG, IORG, and the name of the segment being activated.

- C. IORG is set to the IORG value in the segment descriptor and FREG is set to the VS index of the function mark.
- D. An entry is pushed to LS for the segment with FBASE set to FVBASE, and the process is complete.

The DOI instruction is interpreted the same as the DC instruction unless the top entry in the VS has the tag NPT, in which case, the value referenced is copied to a new space in M. If the tag is SGT, temporary space is allocated for the result and the segment is evaluated. And the segment is evaluated. The top entry of the VS contains the tags ST, JT, or FDT after a DOI instruction is interpreted.

A RETURN instruction is interpreted in the following manner.

- A. The LS is popped, thereby de-activating the segment.
- B. The segment name, encoded in the function mark on the VS, is used to access the segment descriptor and then popped. The value is pushed to the VS, and its NT entry is erased if there is a result. All other NT entries for variables local to the program segment, and their values, are erased.
- C. FREG and IORG are restored from the values in the



function mark on VS, which is then deleted, and replaced by the result, if any.

- D. FBASE is set to point to the current active segment, if any, by accessing its segment descriptor in the newly-exposed function mark.

### 5.3.3 Scalar Operators.

The DM expects the VS to have one or two entries for the monadic or dyadic APL scalar arithmetic operators, respectively. If the two top entries contain scalar values, the operation is done immediately, leaving the result in the VS after popping both operands. That is, the EM is activated immediately to operate on the VS. If one VS entry contains a scalar value, it is entered into the QS and its VS entry is replaced by an appropriate segment descriptor. This case becomes the case of two segment descriptors in VS.

When the DM encounters two segment descriptor entries, it compares the ranks and dimensions and type of the two operands and signals an error if they are not conformable. Otherwise, the operation is deferred in QS by drag-along, and the top of the VS is adjusted so that it contains a segment descriptor pointing to the entire deferred expression in the QS. The deferred code for both operands will always be contiguous in QS because of the stack



discipline of APIM. The LINK field of the QS entry for the operator is a relative backwards pointer to the earliest deferred operand in the deferred subexpression. The AUX field is the same as that of the two operands.

#### 5.3.4 Selection Operators.

The selection operators are evaluated in the EM by the process of beating, as explained in Chapter Two. Briefly however, the array-valued operand is checked for conformability. If the operand is conformable and beatable, it is beaten according to the transformation shown in Chapter Two. In the process, the DA is transformed directly if the reference count is 1. Otherwise, a copy of the DA is beaten. If the result of beating is a scalar value, the EM is called to execute the segment of code in the QS, and leave the scalar result as the top entry of the VS. If the operand of a selection operator is not beatable, it must be evaluated by the EM and a temporary value must be stored. The temporary value is then beaten.

#### 5.3.5 Non-deferrable Operators.

This class of DM operators contains those API operators which cannot be conveniently deferred, or must be evaluated immediately by the DM because they are impossible to defer.





The process of beating may not be applied to those operators because only deferrable operations may be beaten. Therefore, they are executed in a manner similar to present implementations of APL. The DM evaluates a DECODE instruction and pushes the scalar result to the VS with the tag ST. REP, GDU, and GDD are evaluated immediately by the DM because they require complex calculations involving their entire operands simultaneously. These calculations may be impossible or very difficult to do element-by-element. Since the execution of URHO requires fetching the dimensions of its operand, it is easily done by the DM and is therefore not deferred. Evaluation of UICTA is done by the DM by creating the appropriate j-vector. The evaluation of the CAT K instruction causes the DM to concatenate the top K values in the VS and push the resulting array to temporary space in M. These operations are executed as in present implementations and therefore are not included in this thesis.

It is difficult for EM to evaluate the APL operators RAV and DRHO because arbitrary elements of the result may be accessed. However, DM can defer execution in the following cases.

- A. The right-hand operand of RAV and DRHO is a scalar or a single-element array. The RAV of the operand is either





a j-vector or an explicit one-element vector. DRHO of such a value may be deferred in QS as a load scalar (S) instruction followed by an IRD instruction. IRD contains the dimension of the result in the DA it describes, and is used only by the DM.

- B. The right-hand operand is an expression deferred in the form of A above. The DA pointer is made to point to the DA referenced by the deferred expression.
- C. The right-hand operand is an IFA instruction which has not been altered by any selection operations which change the order of the value part.

If none of the above situations apply, then RAV and DRHO are evaluated immediately by the DM and the results are pushed to temporary space in M.

#### 5.3.6 Subscript Operator.

Upon encountering the SUBS K instruction, the DM expects the VS to contain a QS segment descriptor for a rank-K structure, and the next K entries to be either scalars or QS segment descriptors. Interpretation of the SUBS K instruction depends on the condition of the subscriptee.

- A. Subscriptee is beatable. See section 2.3 for methods of beating. The subscriptee expressions are examined

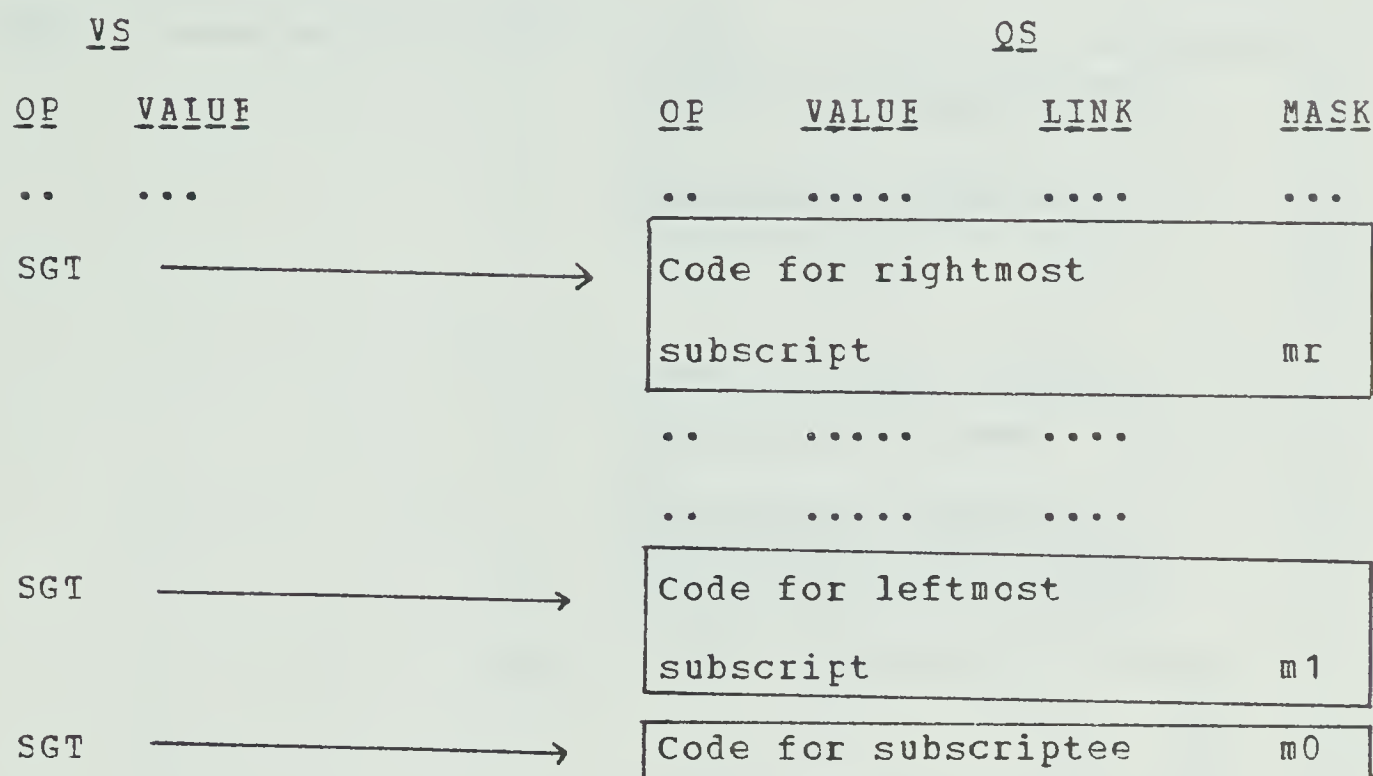


in turn, from the rightmost (deepest in VS), to find scalar values or j-vectors. If found, for example, in coordinate I, the equivalent of the INX I instruction is performed on the subscriptee by beating. After the new DA's have been created by the beating process, the VS entry for this subscript is deleted if it was a scalar. If the VS entry was a j-vector, it is changed to the empty segment and the QS entry is deleted by moving all the QS entries above, one entry toward the QS base, with appropriate adjustments to the descriptors. If the remaining stacked subscriptors are empty or non-existent after all subscripts have been examined, the result already exists in the QS. Then the remaining empty segment descriptors are removed from VS and the result is the QS descriptor at the top of the VS. Otherwise, the remaining subscripts are treated as non-beatable.

- B. Subscriptee is Non-beatable. If the subscript expression is explicit non-scalar or non-j-vector, or the subscriptee is not beatable, it must be deferred in the QS. This is done by creating temporary index accumulators (operator XT) in QS and generating EM code to activate the subscript evaluation at the right times. The QS contains the following code before the

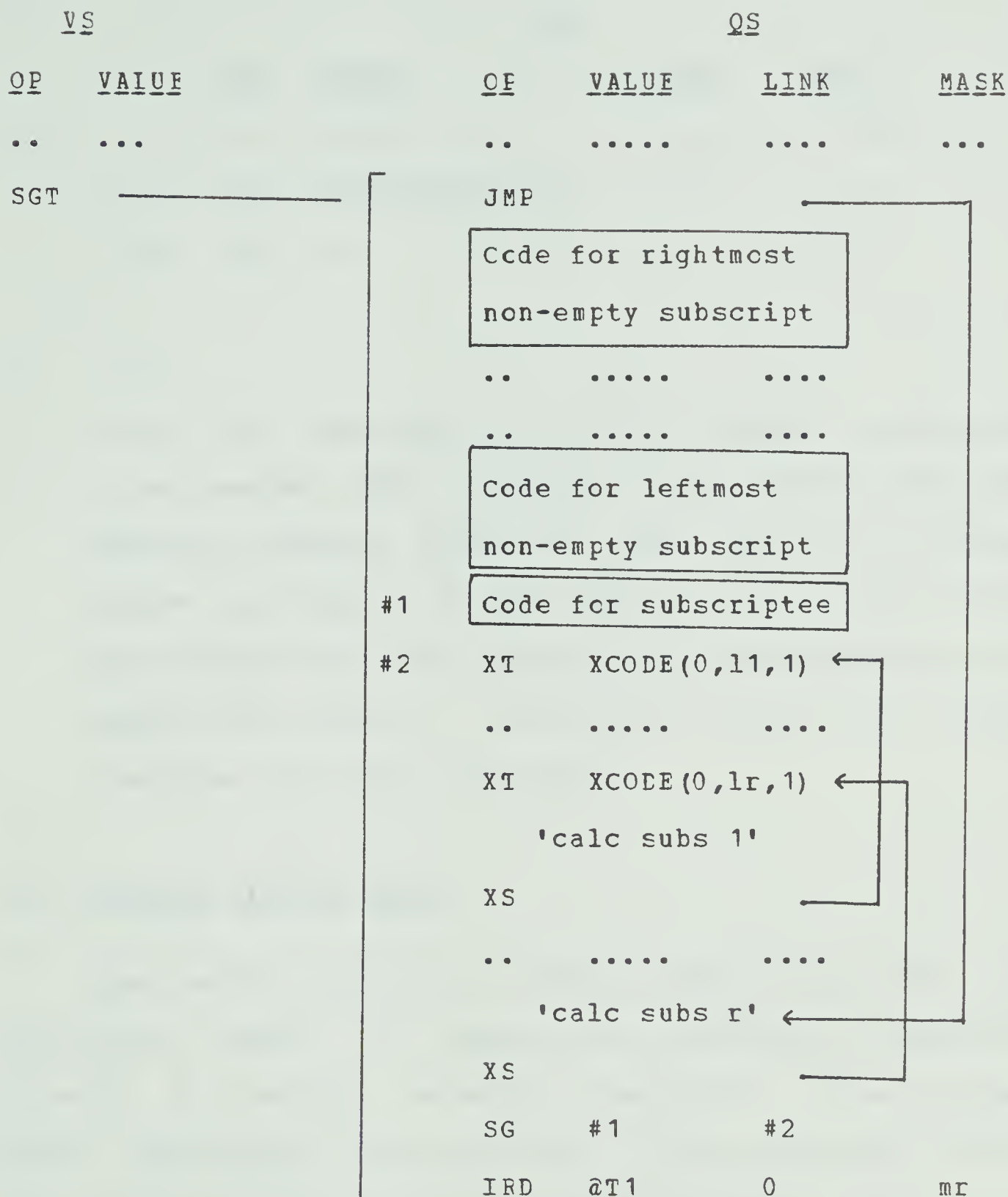


expansion of the subscript expression, where  $r$  is the rank of the subscriptee. The mask is denoted by the character  $m$ .



After expansion the QS contains the following code where  $L_1, L_2, L_3, \dots, L_{r-1}, L_r$  are the dimensions, minus 1, of the subscriptees. #2 is the QS index of the beginning of the XT instructions and @T1 points to the DA which contains the rank and dimensions of the result. The access mask of the result is  $m_r$ , and the rank of the subscriptee is used in the initialization of IA, IFA, and IJ instructions.





The instructions 'calc subs K' are either of the following.





QS

	<u>CP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
	..	.....	....	...
(A)	ISC	SCCDE (SEG.k', 1)	0	m'
(B)	IXI	0	0	m'

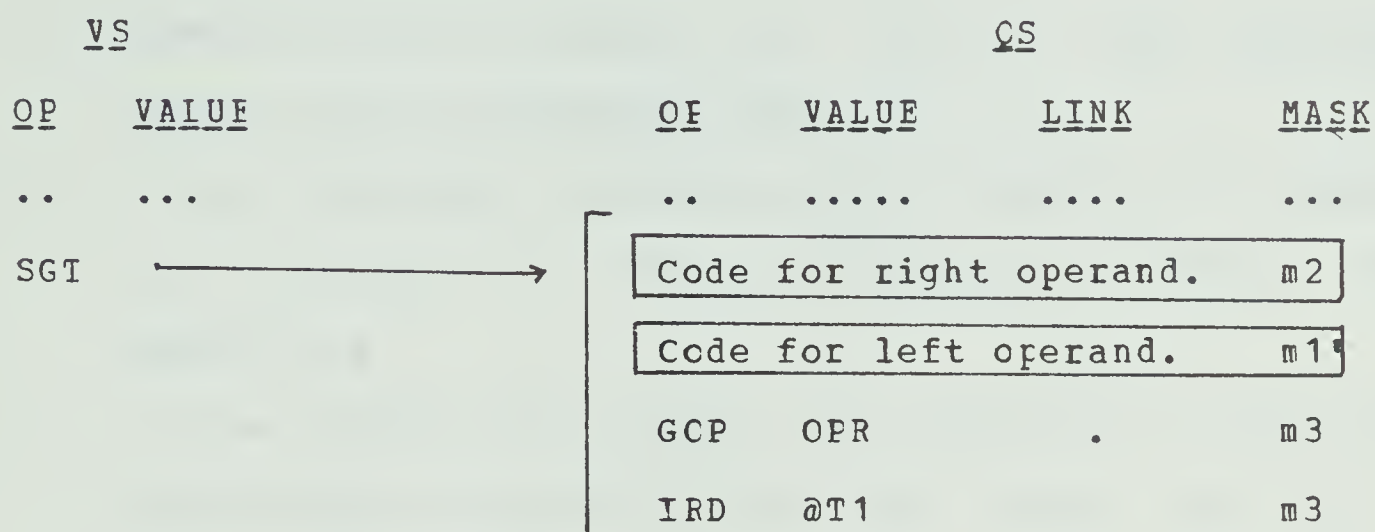
in case (A), subscript K is to be computed explicitly by activating SEG K' which is one of the non-empty subscript segments is QS. In case (B), the stacked segment was empty, so the actual subscript used is the same as that which was controlling this coordinate from outside this segment. The mask m' in the AUX field specifies the index environment.

### 5.3.7 General Dyadic Form.

When the DM encounters a general dyadic form (GDF), the right-hand operand is examined and evaluated to temporary space if it contains deferred operators. This process avoids unnecessary re-evaluation of sub-expressions which are needed to compute a GDF, and guarantees the possibility of applying standard transforms to GDF expressions by beating. All that remains to be done is to alter the access masks in the MASK fields of the deferred left-hand operand



in QS to provide the proper access environment for the EM. If one of the operands is a scalar, the GDF is treated as a normal scalar operator expression. If neither operand is a scalar, the expanded EM code is as follows.



OPR is the operator in the GDF and @T1 points to the DA containing the rank and dimensions of the result. Masks for the operands and the GOF instructions are m2, m1' and m3, where m1' is m2 shifted left by the rank of the right operand, and m3 is the logical 'or' of m1' and m2. The segments must be either a single IJ or IFA instruction because the operands may only be simple array values.

#### 5.3.8 Reduction Operator.

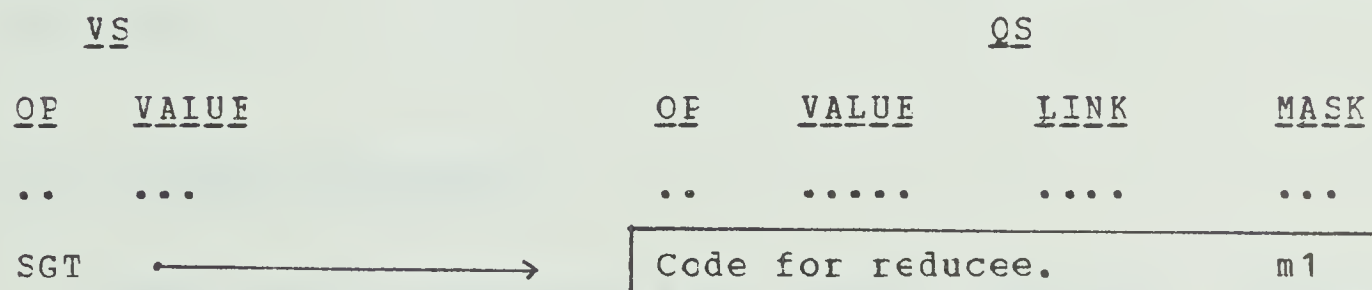
When the RED K CF instruction is encountered by the DM, it examines the reducee for conformability to several



special cases. The cases and their interpretations are as follows:

- A. The reduction coordinate is empty, then the result is an array with the value  $((K \neq 1) \rho \rho R) / \rho R) \rho IDENT$  where R is the reducee and IDENT is the identity element for the reduction operator.
- B. If the reduction coordinate is of length 1, the result is  $R[[K]IORG]$ . If the reducee is a scalar, the result is R.
- C. If the reducee is a vector, the reduction is activated immediately in the EM because the result will be a scalar.

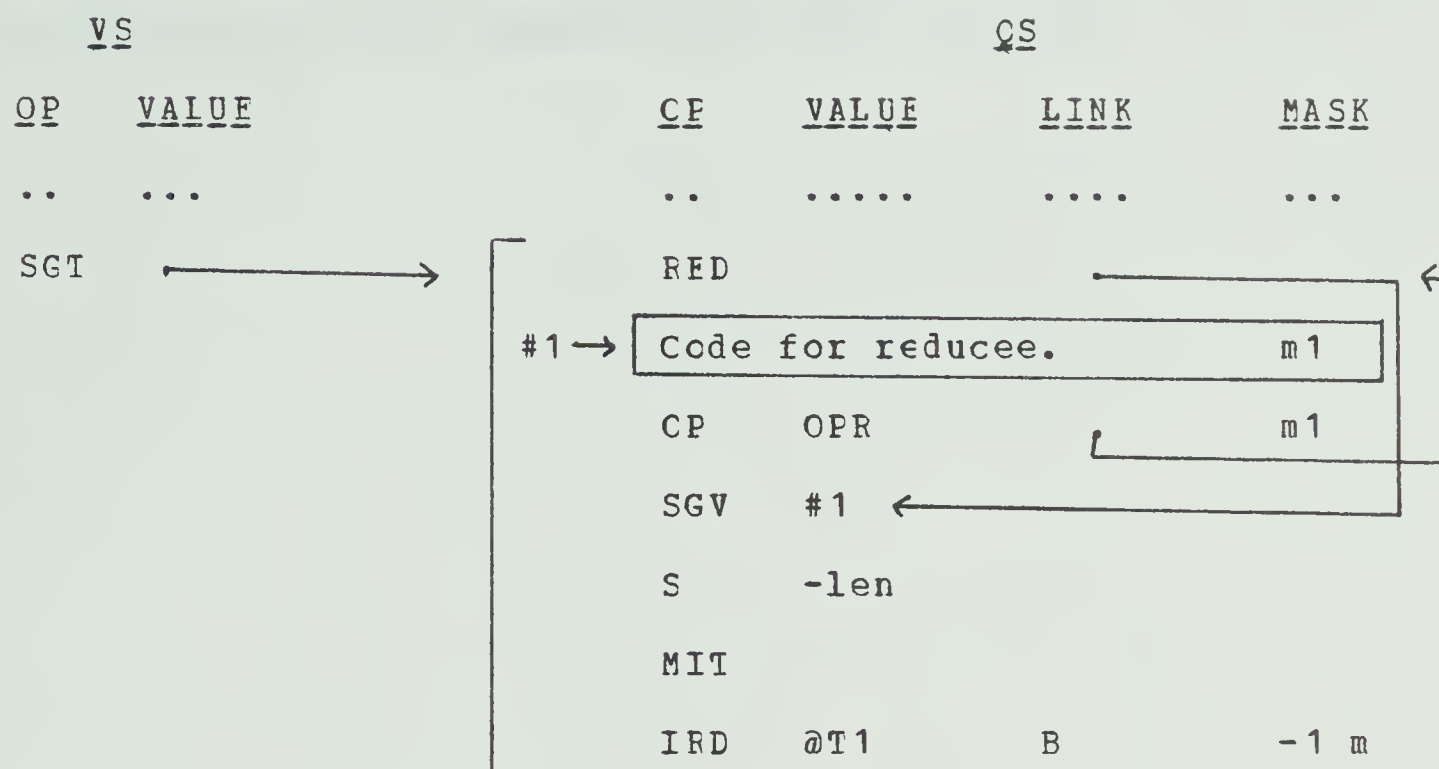
If none of these special cases is found, the reductions is deferred by first performing the transpose, if necessary. The QS would appear as follows, before deferment in QS.



After the deferral process, the QS would appear as follows,



where OP is the reduction operator.



the mask *m* describes the rank of the result and the mask *m1* describes the mask *cf* of the operand. @T1 points to the DA which contains the rank and dimensions of the resulting array. B has the value 1 if the reducee is beatable, otherwise 0.

### 5.3.9 Ranking Operator.

When the DM encounters the ranking operator (DIOTA), the left argument is expected to be a simple vector array value, and is evaluated immediately. If the rank of the





result is 0, i.e. The right operand is a scalar, the following expansion in the QS is executed immediately by the EM, where len is the length of the left operand.



<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	...	..	.....	.....	...

SGT

#1

Code for right operand. m2

JMP

Code for left operand. m1

DUP 2

OP NE

JN1 3

PCP

IVE

OP ADD

ORG

SGV #1

S len

MIT

VXC

PCP

IRF



### 5.3.10 Rotation Operator.

The rotation operation (ROT K), is a special case of subscripting, defined as follows: If N is a scalar, then

$R \leftarrow \phi[K]M$  means for each  $L \in \text{ELT } 1:\rho M$

$R[; /L] \leftarrow M[; /((K-1) \uparrow L), (IORG + (\rho M)[K] | (N - IORG) + 1(\rho M)[K]), K \downarrow L]$

where  $\text{ELT}$  is a function defining occurrences of the left operand vector as one of the rows of the right operand matrix, IORG is the lower bound on the subscripts.

If N is an integer array with  $\rho N \leftrightarrow (K \neq 1:\rho \rho M) / \rho M$  then

$R[; /L] \leftarrow M[; /((K-1) \uparrow L), (IORG + (\rho M)[K] | (N[; /L'] - IORG) + (\rho M)[K]), K \downarrow L]$

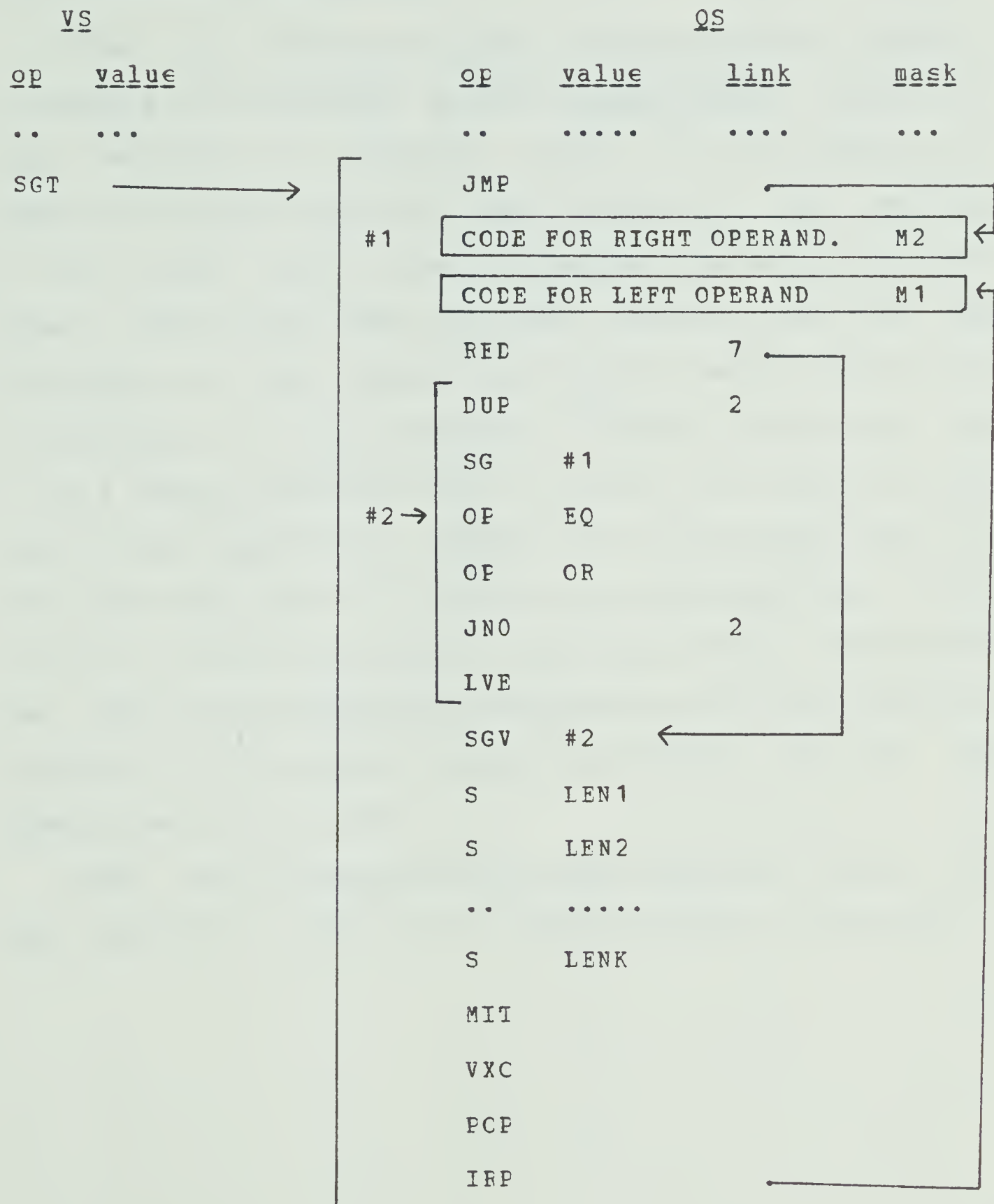
where  $L' \leftarrow (K \neq 1:\rho \rho M) / L$ . The general SUBS K with all but coordinate K being IXL, and XS pairs and the coordinate K being computed according to the above definition. The explicit expansion will be omitted since it is similar to what has already been shown.

### 5.3.11 Membership Operator.

When the LM encounters the membership operator (EPS), it examines the operands for several special cases. If the right operand or the left operand is a scalar, the result is the evaluation of the logical 'equal' operator. If the right operand is a single element quantity, the logical 'equal' operator is evaluated using the left operand, and the first element of the right operand. In all other cases,



EM code is pushed to the QS in the following form.





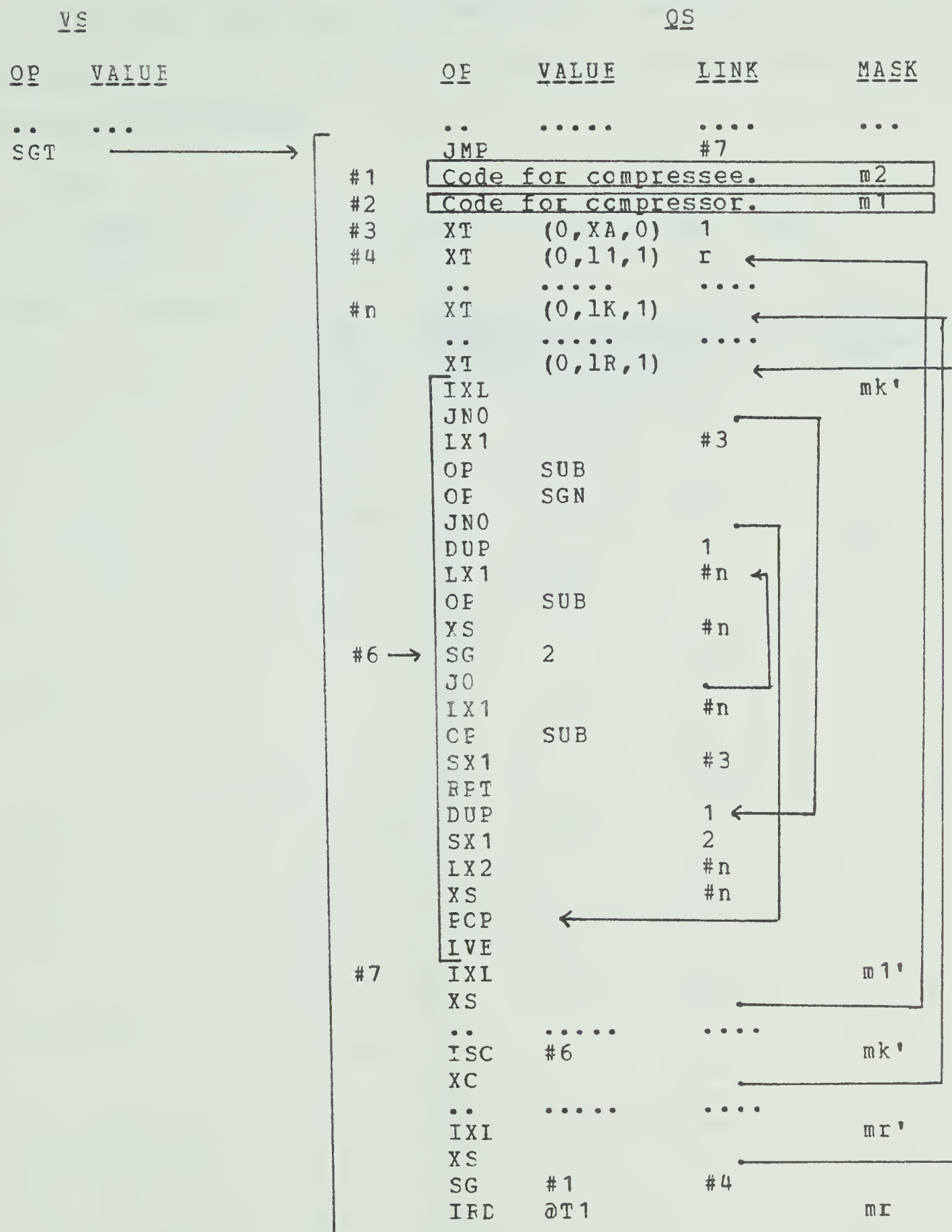


### 5.3.12 CCMPRESS AND EXPAND.

When the DM encounters the compress operator (CMPRS), it expects the compressor to be a logical vector which has been evaluated to temporary space. It also expects the number of logical 1's, DIM1, and the index of the position of the first non-0 value in the left operand, XA, to be known. These values must be known because the rank and dimensions of the result must be known before deferral of the operator. If the compressor is either a scalar of 1 or 0, or a vector containing all 1's or all 0's, the evaluation may be done immediately. However, all other cases result in EM code being pushed to the QS in the following form, where  $l_1, l_2, \dots, l_r$  are the masks of the individual subscripts. The mask for the compressed coordinate is  $mK'$ , and XCODE is bracketed. @@T1 points to the DA containing the rank and dimensions of the result.

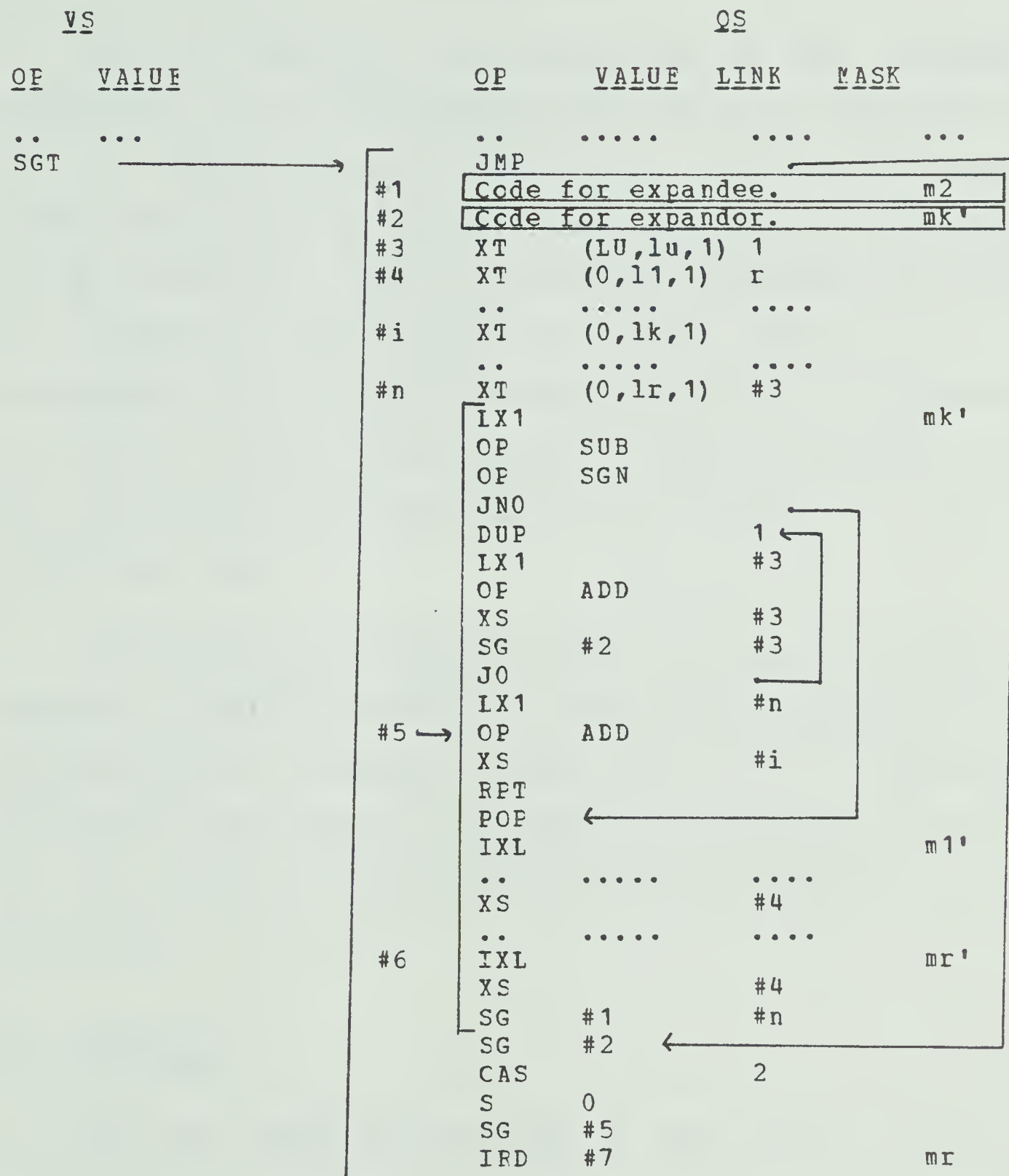
When the DM encounters the expand operator (EXPND), the same situation as that for the CMPRS operator is expected.







The EXEND operator is evaluated immediately if the expandor is all 1's or 0's, otherwise EM instructions are generated as follows.





## The E-Machine

### Chapter Six

The EM performs the evaluations on the operands according to the instructions which the DM has generated in the QS. The LS is used by the EM to provide information about the location and length of the segment in the QS which is to be executed. The IS provides the necessary values for an element-by-element execution process. The VS acts as an accumulator or as temporary storage in that all evaluations are made on data in the VS and the result is placed on the VS. This very briefly describes the function of the EM within the AFIM.

This chapter is written so that the reader may acquire a workable knowledge of the EM. More detailed information is available from a companion thesis by W. F. Appleyard. Therefore, this chapter is designed to explain the operation of the EM by examples and by written explanations of its functions.

#### 6.1 Examples.

The most common instructions, S, IFA, FA, IA, A, IJ, J, and OP, cause the values of operands to be pushed to, popped





from or changed in the VS. Two examples will illustrate their use.

Example 1:  $SUM \leftarrow 5 + 7 + 8$

<u>Opcode</u>	<u>Coperand(s)</u>	<u>Comments</u>
S	8	Push scalar value 8 to VS
S	7	Push scalar value 7 to VS
GCP	ADD	7 to 8 and pop VS
S	5	Push scalar value 5 to VS
GOP	ADD	Add 5 to 15 and pop VS
IA	@SUM	Push address of SUM to VS
OP	ASGN	Store 20 in SUM and pop 2 elements off the VS

The expression results in the generation of the above code in the QS. Load Scalar, S, pushes the constant to the value stack. GCP takes as its operand one of the dyadic scalar arithmetic operators. OP takes as its operand one of the monadic scalar operators. The operand of GOP is applied to the two top elements on the VS, and the result replaces the operands on the VS. ASGN is treated as a monadic operator because the assignment is done to the right hand operand.



Example 2:  $A \leftarrow B - C$

<u>Opcode</u>	<u>Operand(s)</u>	<u>Comments</u>
IFA	@C, MASK=3	Initialize Load Array Element
IFA	@B, MASK=3	Initialize Load Array Element
Op	SUB	Apply subtract operation to VS
IA	@A, MASK=3	Initialize Load Array Address
OP	ASGN	Store VS value

Assume that B and C are conformable for the add operation, and are ten by ten matrices. After the LS overflows for the first time, that is, the section of code shown has been executed for one element of each array, the QS becomes:



<u>Opcode</u>	<u>Operand(s)</u>	<u>Comments</u>
FA	LINK=5,VBASE=100,SUM=10	
FA	LINK=6,VBASE=150,SUM=8	
OP	SUB	
A	LINK=7,VBASE=325,SUM=0	
OP	ASGN	
NT	Q1=40,Q2=10,INX=0	ICB for C
NLT	Q1=8,Q2=1,INX=1	
NT	Q1=36,Q2=9,INX=0	ICB for B
NIT	Q1=-8,Q2=-1,INX=1	
NT	Q1=36,Q2=9,INX=0	ICB for A
NLT	Q1=8,Q2=1,INX=1	

The IFA instruction performs the same function for arrays as S does for scalar values. The IFA is used to initialize a Fetch Array Element instruction, FA, and to link the FA to an Iteration Control Block, ICB, at the top of the stack of instructions. ICB's are explained in the next section. An FA instruction causes array elements to be pushed to the stack. The ICB's control the indexing of array elements. The indexing mechanism of the EM is explained in section 6.3. The FA instructions push the values at C[1;1] and B[1;1] to the VS. The GOP instruction performs the subtraction and replaces the operands with the



result. The IA instruction is used to initialize a Load Array Address, IA, which is linked to an ICB entry. The A instruction pushes the memory address of A[1;1] to the stack. The following instruction stores the result of the subtraction at the memory location of A[1;1]. After the assignment of A[1;1], the IS overflows. When this condition occurs, the IS is stepped and the LS set to re-execute the segment of code for the elements C[1;2], B[1;2] and A[1;2]. This process continues until the iteration has covered the range of values represented by the IS.

The EM instructions IJ and J access j-vectors and have the following formats.

<u>Opcode</u>	<u>Coperand(s)</u>
IJ	JCODE (J1,J2,J3) ,MASK
J	XCCDE (X1,X2,X3) ,INX

J1 is the length, or the number of values, to be generated by this instruction. J2 is the starting or ending value of the vector, and J3 specifies whether the generated values should start from or decrement to J2. X1 is the current value. X2 is the starting value. The value defined by the instruction is  $X2 + X1$  depending on the value of X3. The operands MASK and INX allow J to be dependent on the index environment as explained in the next section.





## 6.2 Indexing Non-subscripted Variables.

This section explains in detail the mechanisms which facilitate the element-by-element execution of a segment of EM instructions in the QS. The IS is used to index the segment in the normal sequence and the IS specifies the elements upon which evaluations are to be made. Recall that an LS entry contains the starting address of a segment of code, ORG, the length, LEN, and a counter, PEI, which acts as an index. The relevant flags are LE for machine control, and IS for iteration involvement. Also recall that an IS entry contains a counter, CTR, and a maximum value, MAX, which represents the range of that particular rank represented by the entry. The relevant flags are DIR, for decrementing the CTR, CH, for recording the changing of the IS entry, and MRF, for marking the last level of indices pushed to the IS.

The value of CTR in the IS is used as an argument in the storage access function to access the individual elements in an array. Each time the segment of EM code is executed, the IS is stepped for the next element-by-element evaluation of the segment. The IS is stepped by incrementing CTR at the top of the IS and marking the CH bit. If CTR does not overflow, that is CTR is less than or



equal to MAX, the stepping process is complete. If CTR does over-flow, it is reset to zero and the process repeated for the next CTR lower in the stack. However, if the MRK bit is on in an entry that overflows, the stepping of the IS is terminated, and the IS popped. This condition occurs when all the elements of the array have been accessed that were specified by the range of the IS. The action of the IS is analogous to a set of nested loops. The inner-most loop, the loop incrementing along the rows, is at the top of the IS. The dimensions of the highest rank are located deepest in the IS. The outer most loop is marked with the MRK bit. The highest rank that can be handled is 32.

The function that the ICB's perform is the efficient calculation of the storage access function. ICB's are created in the process of initializing an FA from an IFA instruction. An IFA instruction has a 32 bit mask, the maximum number of entries in the IS, as an operand. A bit turned on in this mask determines the creation of an entry in the ICB group. The right-most bit corresponds to the top of the IS. In example 2, the mask selected the top two entries of the IS. In each case the array involved is of rank 2. The position of the bit in the mask determines the value of INX in the ICB. The INX value is the displacement from the base of the IS that corresponds to the ICB entry.



A zero bit in the mask eliminates that co-ordinate from the indexing of the array.

The first operand of an FA instruction is a link to the ICB group at the top of the QS and the second operand is the location of the array in memory. The third operand, SUM, corresponds to the value of the storage access function. The ICB entries are used to alter SUM each time an FA or A instruction is executed. The link in the FA or A instruction is used to locate the ICB group. Entries in the group are denoted by the tag NT, and the last entry has the tag NLT. The INX value in each ICB entry selects an entry in the IS. If the selected entry in the IS has the CH bit off, the next ICB entry is examined. If the CH bit is on, the CTR was changed when the IS was last stepped and the corresponding ICB entry is used in the calculation of SUM. Usually it is only necessary to use the ICB entry corresponding to the top of the IS in the calculation of SUM. It is only when the entry at the top of the IS overflows that the other entries in the IS are stepped.

ICB's have entries labeled Q1 and Q2. If an IS entry is incremented on stepping, the Q2 value is added to SUM. If the IS entry overflows, the Q1 value is subtracted. The value of Q2 is such that when added to SUM it will change SUM to be the displacement of the next column, row, plane,





etc. The value of Q1 will reset sum to the beginning of the coordinate. That is, if SUM has incremented across the columns in a row, subtracting Q1 from SUM will point SUM to the first column in that row. Q2 is the number of elements along a dimension of an array. When an entry in the IS overflows, Q1 is used to reset SUM.

The IS performs the same functions for the APIM that a set of nested loops provides for a program written in FORTRAN. However, the evaluation of the storage access function is optimized because of the regularity in the stepping procedure of the IS.

### 6.3 Indexing Subscripted Variables.

The ease with which the IS can be stepped through a range of indices makes it a very efficient method for generating the arguments for storage access functions. The values of indices for subscripted variables must be fetched from memory and placed in the CTR field of a pseudo-iteration stack. The pseudo-iteration stack is kept in the QS because one is needed for each subscripted variable or expression in a QS segment. The fields in the pseudo-IS are labeled X1, X2, X3, which are equivalent to the CTR, MAX, and CH fields of the IS.

The pseudo-IS is stepped in a manner similar to the IS.





Where possible, arguments for the storage access function are derived from the IS. The EM instructions, Index Load and Index Store, XL and XS, are used to transfer the value of the CTR fields from the IS to the pseudo-IS. The Activate Segment Conditional instruction, ISC, is initialized to an SC instruction which pushes an entry to the LS which activates the EM. The segment of code causes a value to be pushed to the VS. An XS instruction is then used to push this value to the X1 field. A recursive method for handling nested subscripts exists because the expression may be indexed. The SC is conditional in that it tests the CH field to determine if the X1 field of the pseudo-IS requires changing. Example 3 illustrates the function of the IS.

Example 3: A[I[J;];]

<u>Operation</u>	<u>Operand(s)</u>	<u>Comments</u>
00 JMP	13	
01 IFA	@J, MASK=4	
02 IFA	@I, MASK=6	
03 XT	XCODE(0,4,1), GN=2	GN - Group number of XT
04 XT	XCCDF(0,6,1)	GN ignored for following entries
05 ISC	SCODE(4,1,1), MASK=4	Invokes IFA @J segment
06 XS	LINK=3	Link points back to XT entry at 03
07 IXL	MASK=2	Mask causes IXL to access



```

08  XS          IINK=4          CTR second from the top
                                Link points back to XT
                                entry at 04
09  SG          SCODE(7,1,1),LINK=6 Invokes IFA @I
                                segment
10  IFA          @A,MASK=7
11  XT          XCCDE(0,8,1),GN=2
12  XT          XCCDE(0,10,1)
13  ISC          SCCDE(8,5,1),MASK=2 Invokes segment
                                entries 05 through 09
14  XS          LINK=3          Link point at XT entry 11
15  IXL          MASK=1
16  XS          IINK=4
17  SG          SCODE(7,1,1),LINK=6 Invokes IFA @A
                                segment

```

The first pass through the code modifies it to the following form:

<u>Operation</u>	<u>Operands(s)</u>	<u>Comments</u>
00	JMP	13
01	FA	LINK=29,VBASE=100,SUM=0
02	FA	LINK=29,VBASE=200,SUM=4
03	XT	XCODE(4,4,1),GN=2
04	XT	XCCDE(0,6,1)
05	SC	SCODE(4,1,1),LINK=0
06	XS	LINK=3
07	XL	LINK=1
08	XS	IINK=4
09	SG	SCODE(7,1,1),LINK=6
10	FA	LINK=23,VBASE=250,SUM=0
11	XT	XCCDE(2,8,1),GN=2
12	XT	XCODE(0,10,1)
13	SC	SCODE(8,5,1),LINK=1
14	XS	IINK=3
15	XL	LINK=2
16	XS	LINK=4
17	SG	SCODE(7,1,1),LINK=6
.		
.	Code to excute the rest of the segment from	
.	Which this example is taken.	
.		
30	NIT	Q1=4,Q2=1,INX=0
31	NT	Q1=28,Q2=7,INX=0



```

32  NLT          Q1=6,Q2=1,INX=1
33  NT          Q1=88,Q2=11,INX=1
34  NLT          Q1=10,Q2=1,INX=2

```

This APL sub-expression in some unspecified expression caused the above expansion of code in the QS. The ranks of A, I, and J are 9 11, 5 7, and 2, respectively. The initial values contained in IS and IS are:

	<u>REL</u>	<u>ORG</u>	<u>FLAGS</u>	<u>LEN</u>	<u>OP</u>
Top of IS	0	0	110	17	11

	<u>CTR</u>	<u>FLAGS</u>	<u>MAX</u>
	0	011	1
	0	010	6
Top of IS	0	010	10

The IS initiates the segment of code between locations 0 and 17. The JMP instruction at location 0 modifies the IS REL field to value 13. The ISC instruction at location 13 is initialized to SC which pushes to the IS an entry invoking the QS code to evaluate the subscript.

The following IS entry invokes execution of the instructions at locations 5 through 9.

	<u>REL</u>	<u>ORG</u>	<u>FLAGS</u>	<u>LEN</u>	<u>OP</u>
	14	0	110	17	11
Top of IS	0	5	1101	5	3



the QP field is set to indicate the pseudo-IS for the invoked segment. The ISC instruction at location 5 initializes and pushes to the LS, code to evaluate the inner subscript.

The QS segment described by the following LS entry is the IFA instruction at location 1.

	<u>REL</u>	<u>ORG</u>	<u>FLAGS</u>	<u>LEN</u>	<u>QP</u>
	0	0	110	17	11
	1	5	1001	5	3
Top of IS	0	1	1001	1	0

this instruction pushes the first value of J to the VS. The LS overflows and is popped if its IS field indicates that the IS does not have control. Execution resumes at location 6. The XS instruction links back to an XT entry and fills in its X1 field. The following XI, XS pair use the IS to fill in the following XT entry. The SG invokes the single instruction at location 2, using the LS. The IFA at location 2 initializes and pushes the first subscripted value to the VS. The top two entries in the LS are then popped because both overflowed and execution resumes at location 14. There is now one entry in the IS. The instructions at locations 14 through 17 are executed in much the same way as those in locations 6 through 9 with the







following exception. When the IS overflows, its IS field indicates that the IS is to be stepped. REL is reset to zero, and the segment re-executed. Modifications to the method of handling subscripted expressions which permit handling of non-subscripted expressions are a mechanism to fetch values from subscripts and the addition of pseudo-IS stacks to contain them.

#### 6.4 Control Transfer.

The instructions for transferring control are the jump instructions, JMF, J0, J1, JN0, JN1. JMP N is an unconditional jump to an instruction which is N forward from the JUMP. 0 or 1 in the other instructions indicate that the instruction is conditional on the value of the VS being 0 or 1. J0 and J1 pop the top entry of the VS, as well as execute the jump command.

CY, LVE, RPT and CAS perform the functions of CYCLE, LEAVE, REPEAT and CASE. CY causes the IS to overflow which in turn steps the IS and repeats the segment if the IS does not overflow. LVE simply pops the IS, de-activating the current segment. RPT resets REL to 0. RPT is similar to CY but does not step the IS. CAS N,L invokes a single instruction at displacement N forward from the current location. If the invoked instruction pushes an entry to the



LS then a new segment can be initiated. Execution resumes at displacement L from CAS when the current LS entry is again at the top of the LS.



## Sample Expression

### Chapter Seven

In order to explain more fully the operation of the AFIM, this chapter takes an arbitrary APL expression and shows how the AFIM translates the expression, step-by-step, into machine executable instructions. The following APL expression is used, not because it is a useful algorithm, but because it illustrates application of the basic concepts of the AFIM.

$$F \leftarrow 1 \div E \leftarrow -4 \times (D * 2) \uparrow C \circ . \times B + A$$

It is assumed that the arrays are defined and conformable for the operations applied to them. Section 7.1 describes the process of compiling a program segment for the APL expression. Section 7.2 shows the step by step interpretive procedure of the DM.

#### 7.1 Compilation by CM.

The CM compiles APL programs into program segments which serve as object modules for the DM. This expression is assumed to be a statement in an APL program, therefore, it will be compiled into a series of DM instructions within



a program segment. These instructions will be represented internally by their hexadecimal codes in a string form. As the CM encounters each named array, a search of the name index table (NIT) is initiated. The name of the array is then replaced by a load instruction whose operand is the index value (INX) of the position of the name in NIT. The load instruction is LDNF if the array is to be processed, and LDN if the array is to be assigned values. The monadic operators are replaced by the MCNADIC instruction whose operand is the monadic APL operator. Dyadic operators are treated similarly. Specification operators are replaced by the MCNADIC instruction whose operand is ASGN or ASGNV, a destructive or non-destructive assignment respectively. Scalar values are replaced by the LDS instruction, whose operand is the scalar value. The resulting program segment is follows, where @ARRAY is the INX value of the array ARRAY.

<u>CICCODE</u>	<u>OPERAND</u>
LDNF	@A
LDNF	@B
DYADIC	ADD
LDNF	@C
GLF	MULT
LDS	2





IDNF	@D
DYADIC	POWER
TAKE	
IDS	4
DYADIC	MULT
MONADIC	MINUS
IDN	@E
CF	ASGNV
IDS	1
DYADIC	DIVIDE
IDN	@F
CF	ASGN

## 7.2 Interpretation by DM.

The DM accesses instructions in the program segment, interprets them and generates EM instructions in the QS. Assume that the DM is in the process of interpreting the program segment and all the previous instructions have been executed. The remainder of this section has the following format. Each sub-section states the action of the EM from the previous sub-section to the current sub-section, if any, and also states the current action of the DM. Each sub-



section is preceded by a display of the contents of the VS and QS, if and after control is returned by the EM. Each sub-section is followed by a display of the contents of the VS and QS, after control is passed to the EM by the DM.

### Step 1.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....

The DM encounters the LDNF A and pushes an IFA instruction which initializes the process for accessing the array A. The VS entry is made to point at this first instruction of the new QS segment. The VALUE field points to the DA for A and MASK (#), is a 32-bit vector whose last n bits are 1, where n is the rank of the array A. One entry for each dimension of the array A is pushed to the IS, which is used by the EM.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SCT	—————→	IFA	@A		#A



Step 2.

A procedure similar to step one is used for the instruction LDNF B. When the DYALIC ADD is encountered, the IS entries for the array B are compared to those for array A for ADD conformability. If conformable, the IS entries for the array B are popped, the VS is popped, and a GCP ADD instruction is pushed to the QS.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	—————→	IFA	@A		#A
		IFA	@B		#B
		GCP	ADD		

Step 3.

A procedure similar to step one is followed for the instruction LDNF C. When the GDF MULT is accessed, the following process is applied. First, if either of the top two VS entries contains an ST flag, a load scalar instruction is inserted in the QS and the VS entry is changed to an SGT pointer, pointing to the load scalar instruction in the QS. However, such is not the case.



Secondly, if the left hand operand (LHO) contains a deferred operator, it must be evaluated to temporary space. However, such is not the case. Thirdly, if the right hand operand (RHO) contains a deferred operator, it must be evaluated to temporary space to avoid unnecessary repetitive calculations. Such is the case. An IS entry is generated, and the instruction pointer in the program segment is made to point back at the IDNF C instruction. The VS is popped, the top QS segment is popped, and the EM is called to evaluate the QS segment to temporary space, labelled TEMP.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	→	IFA	@A		#A
		IFA	@B		#B
		GCP	ADD		

Step 4.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	→	IFA	@TEMP		#TEMP





The EM evaluated the top QS segment to temporary space and the above entries are inserted by the procedure which returns control to the DM. The DM pushes an IFA instruction for the LDNF C instruction and then encounters the GDF MULT instruction again. This time, both operands are arrays, i.e. Both segments contain only IFA instructions. The arrays are checked for conformability to the outer product operation. If they are conformable, the mask for the LHC is shifted to the left n bits, where n is the rank of the RHO. A GCP MULT instruction is pushed to the QS and the VS is popped. In this case, the mask for C is denoted as #TEMP'.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	—————→	IFA	@TEMP		#TEMP
		IFA	@C		#TEMP'
		GCP	MULTIPLY		

#### Step 5.

The DM encounters the LDS 2 instruction and pushes the scalar value to the VS with tag ST. The DM also pushes an IFA instruction to the QS, with an SGT pointer in the VS, for the LDNF D instruction.



<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OF</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	—————→	IFA	@TEMP		#TEMP
		IFA	@C		#TEMP'
		GOP	MULTIPLY		
SGT	—————→	S	2		
SGT	—————→	IFA	@D		#D

Step 6.

When the DM encounters the IYALIC POWER instruction, it notices that the RHC contains a scalar value. The DM inserts an S instruction, load scalar, in the QS and changes the VS entry to an SGT pointer. If the segments are conformable for the dyadic operation, the DM pushes a GOP POWER instruction to the QS, and pops the VS.



<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	—————→	IFA	@TEMP		#TEMP
		IFA	@C		#TEMP'
		GCP	MULTIPLY		
SGT	—————→	S	2		
		IFA	@D		#D
		GCP	POWER		

### Step 7.

When the DM encounters the TAKE instruction, it checks if the LHO is an array. If it is not, the EM is called to evaluate the LHO to temporary space, labelled TEMP2. The instruction pcinter to the program segment is set back to point to the TAKE instruction again.

<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	....	....
SGT	—————→	IFA	@TEMP		#TEMP
		IFA	@C		#TEMP'
		GCP	MULTIPLY		
SGT	—————→	IFA	@TEMP2		#TEMP2



Step 8.

The EM evaluated the top segment to TEMP2 and returned control to the DM, setting up the IFA @TEMP2 instruction. The DM then beats the operands in the second segment by applying the transformations, listed in chapter 2, to the DA's. The VS is then popped and the top segment is popped from the QS. The character % symbolizes a beaten IA and the corresponding mask.

<u>VS</u>			<u>QS</u>			
<u>OP</u>	<u>VALUE</u>		<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....		..	.....	....	....
SGT	—————→		IFA	@TEMP%		#TEMP%
			IFA	C%		#C%
			GOP	MULTIPLY		

Step 9.

The DM then encounters the LDS 4 instruction which results in the generation of a VS entry with tag ST and whose value is set to 4. The DYADIC MULT instruction is encountered and a procedure similar to that in 7.2.6 is followed. A GCP MULT instruction is pushed to the QS for the MONADIC MINUS instruction.





<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	.....	.....
SGT	—————→	IFA	@TEMP%		#TEMP%
		IFA	C%		#C%
		GOP	MULTIPLY		
		S	4		
		GOP	MULTIPLY		
		OP	MINUS		
		IA	@E		
		OP	ASGNV		

### Step 10.

The DM encounters the IDN E instruction and pushes as IA instruction to the QS. Space is allocated for the resulting array E, and its descriptor when the ASGNV instruction is accessed. An CF ASGNV instruction is pushed to the QS.



<u>VS</u>		<u>QS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	.....	.....
SGT	→	IFA	@TEMP%		#TEMP%
		IFA	C%		#C%
		GCP	MULTIPLY		
		S	4		
		GOP	MULTIPLY		
		OP	MINUS		
		IA	@E		
		OP	ASGNV		

### Step 11.

When the LDS 1 instruction is encountered and the DYADIC DIVIDE instruction is encountered, a procedure similar to that in 7.2.6 is followed. The LDN F and ASGN instructions result in the same type of QS instructions as in 7.2.10. The configuration of the QS would be as follows after the ASGN instruction has been interpreted.



<u>VS</u>		<u>CS</u>			
<u>OP</u>	<u>VALUE</u>	<u>OP</u>	<u>VALUE</u>	<u>LINK</u>	<u>MASK</u>
..	.....	..	.....	.....	.....
SGT	—————→	IFA	@TEMP%		#TEMP%
		IFA	C%		#C%
		GCP	MULTIPLY		
		OP	MINUS		
		IA	@E		
		OP	ASGNV		
		S	1		
		GOP	DIVIDE		
		IA	@F		
		OP	ASGN		



## APLM Simulation

### Chapter Eight

A simulation of the APLM has been undertaken by the author and W. F. Appleyard. A subset of APL has been implemented on an I.B.M.360/67, in 360 ASSEMBLER, using the Michigan Terminal System. Appleyard has implemented the EM and the author has implemented the DM. The APLM initializes arrays of data and program segments by reading them in because the CM has only been outlined, not implemented. The remaining sections of this chapter contain a brief introduction to the program units, and their flow-charts, where applicable. Each section refers to an appendix in which the programs for that unit are listed.

#### 8.1 Monitor and Initializers.

The supervisor of the APLM is called MCNITOR, and serves to initialize and pass control to the other initializers, the DM, and the EM. All listings described in this section are given in appendix A.

The initializers and their functions are as follows. SPECARR reads in arrays of data and assigns space in memory for them. SPECIS generates an entry in the location stack





which causes the DM to begin interpretation of the program segment whose address was pushed to the location stack. These programs are not pertinent to the concepts of drag-along and heating, and therefore are only listed, not flow-charted.

The principal initializer, as far as the EM is concerned, is DMACCCDE, which allocates space for segment descriptors and reads in their program segments. A simplified flow-chart for DMACCCDE is contained in figure 8.1.

## 8.2 Maincycle.

The program unit which provides the mechanism for passing of control between the DM and the EM, is MAINCYCL. Each pass through this program permits either the DM or the EM to interpret one of its source instructions, and to decide whether or not to pass control to the other. A listing appears in appendix B, and a simplified flow-chart is contained in figure 8.2.

## 8.3 D-Machine.

The principal unit in the simulation of the DM is called DMACHINE. This program accesses DM instructions in the normal accessing sequence, does some basic analysis, and



passes control to the corresponding interpretive procedures for that particular type of instruction. Control is then passed back to IM by the interpretive procedures. The listing appears in appendix C, and a simplified flow-chart is contained in figure 8.3.

#### 8.4 Load Instructions.

The interpretive procedures for the DM instructions LDS, LDSEG, LDJ, and LDN, are embodied in a program labelled LOAD. This program fetches the required value or name and pushes it to the value stack with the proper environment. The listing appears in appendix D.

#### 8.5 Control and Storage Access.

The procedure named LNF provides the access linkage for array operands. LNF stands for 'load name and fetch array value'.

The procedures which implement JMP, JMP0, and JMP1, are combined in a program labelled JMP. These instructions cause changes in the normal instruction accessing sequence.

The listings of programs described in this section are contained in appendix E. The algorithms are straightforward and are therefore not flow-charted.



## 8.6 Scalar Arithmetic Operators.

The interpretation of all monadic and dyadic scalar arithmetic operators is done in a program labelled MDALIC. This program undertakes conformability checks, generation of EM instruction, and decisions about transfer of control. A simplified flow-chart is contained in figure 8.4.

The implementation of outer products is undertaken by a program labelled GDF, for general dyadic form. Conformability checks, and EM calls for evaluation to temporary space are made by GDF. A simplified flow-chart is contained in figure 8.5.

ASGN and ASGNV are implemented by a program labelled ASGN. A simplified flow-chart is contained in figure 8.6.

All the programs described in this section are listed in appendix F.

## 8.7 Selection Operators.

Selection operations are evaluated by the program named SELECT. This program implements TAKE, DROP, REV, and TRANS, by beating the operands of these operators. A simplified flow-chart is contained in figure 8.7, and a listing is contained in appendix G.

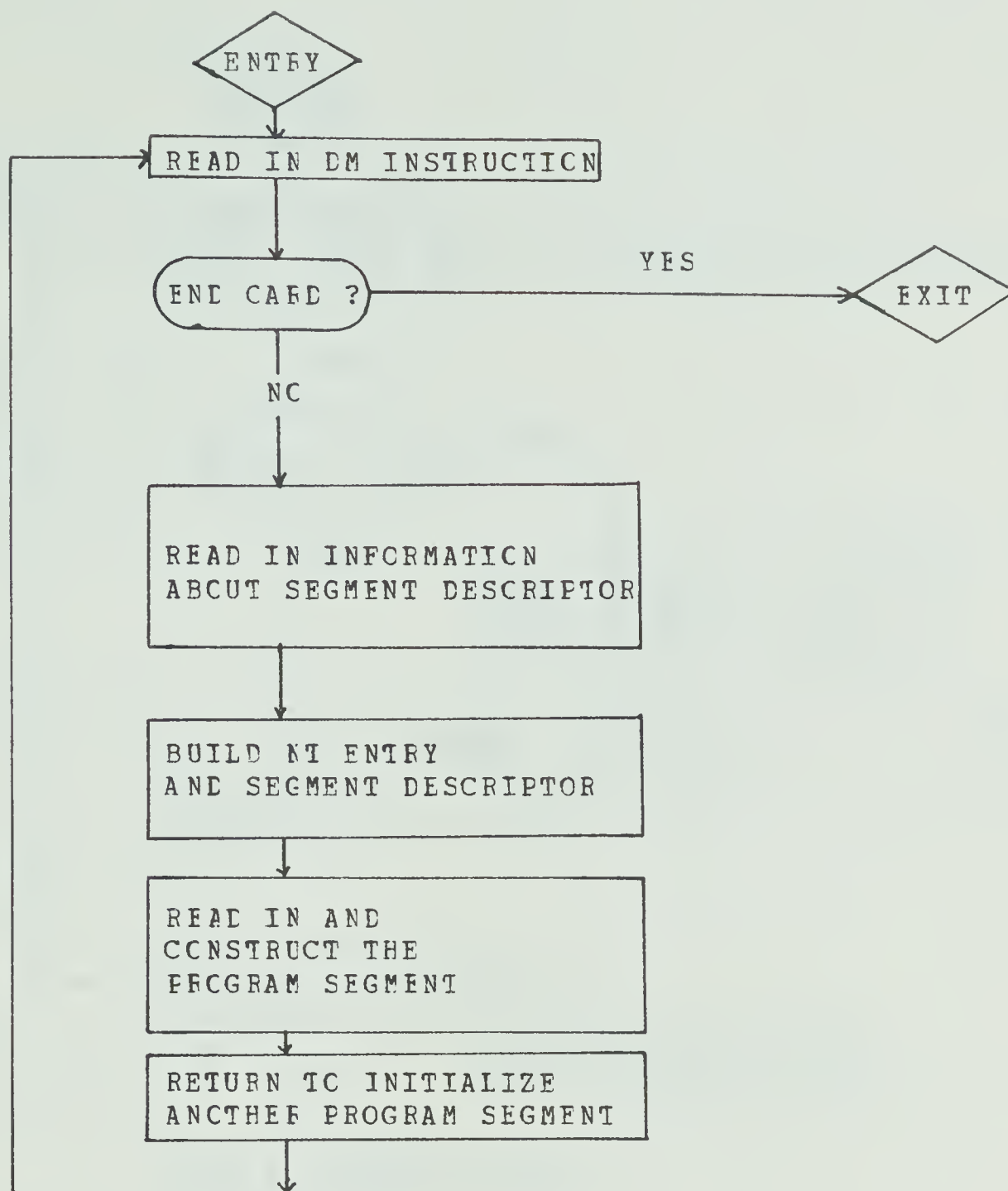


## 8.8 Program Constant.

Appendix H contains a listin og the macros, equates, and all the constants used in the simulation of the DM by the author, and the EM by Appleyard.



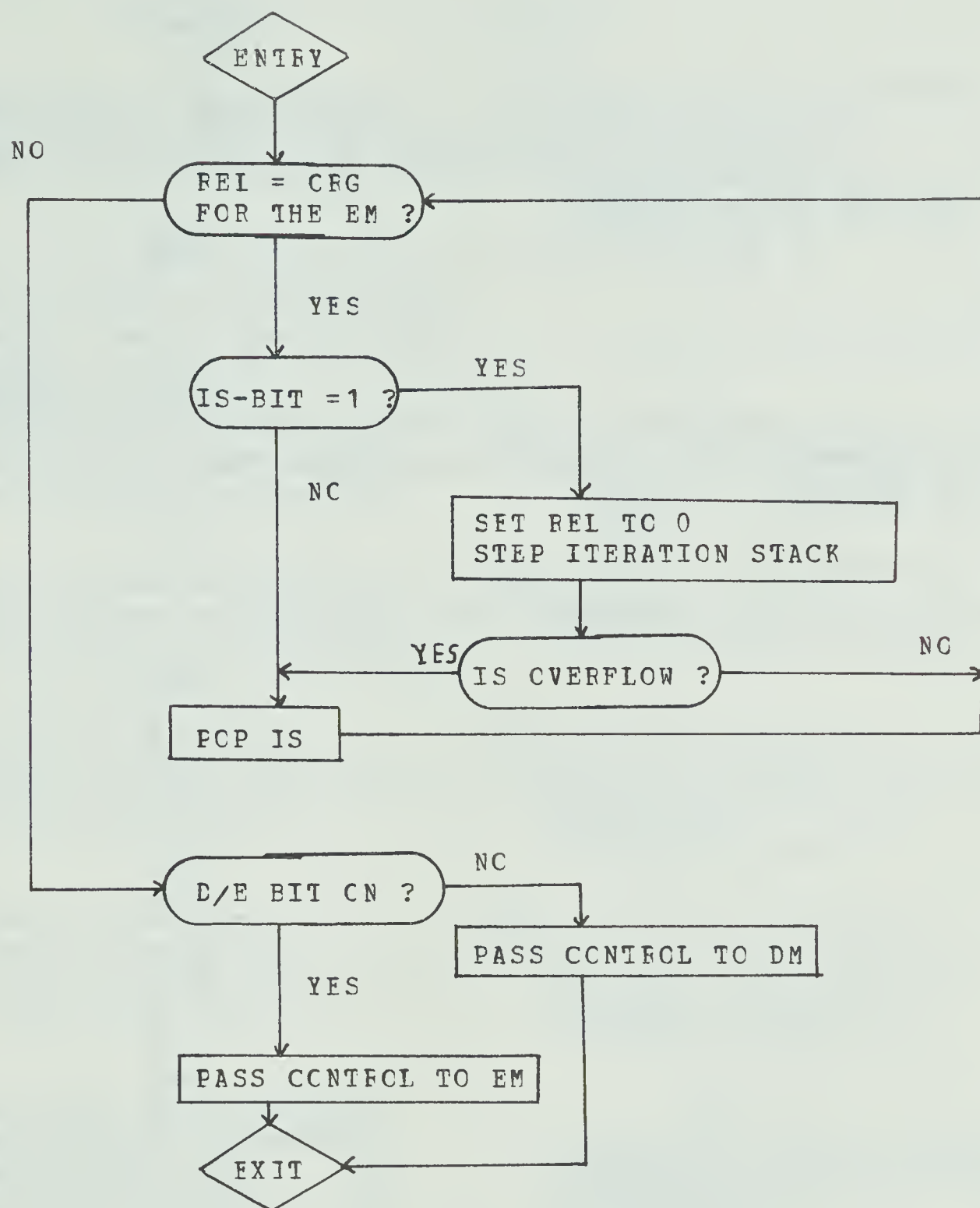




DMACCODE FLOW-CHART

Figure 8.1

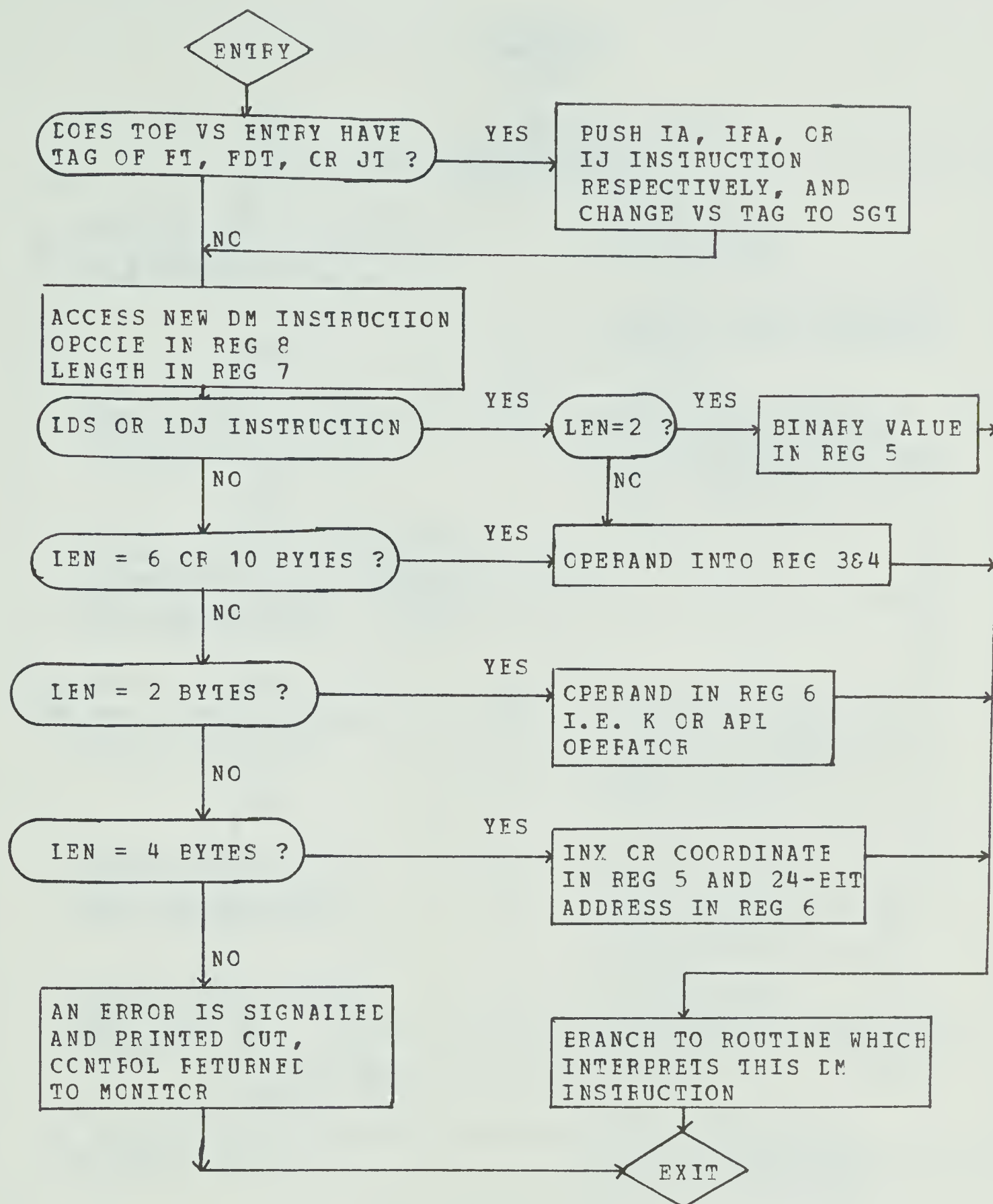




MAINCYCL FLOW-CHART

Figure 8.2

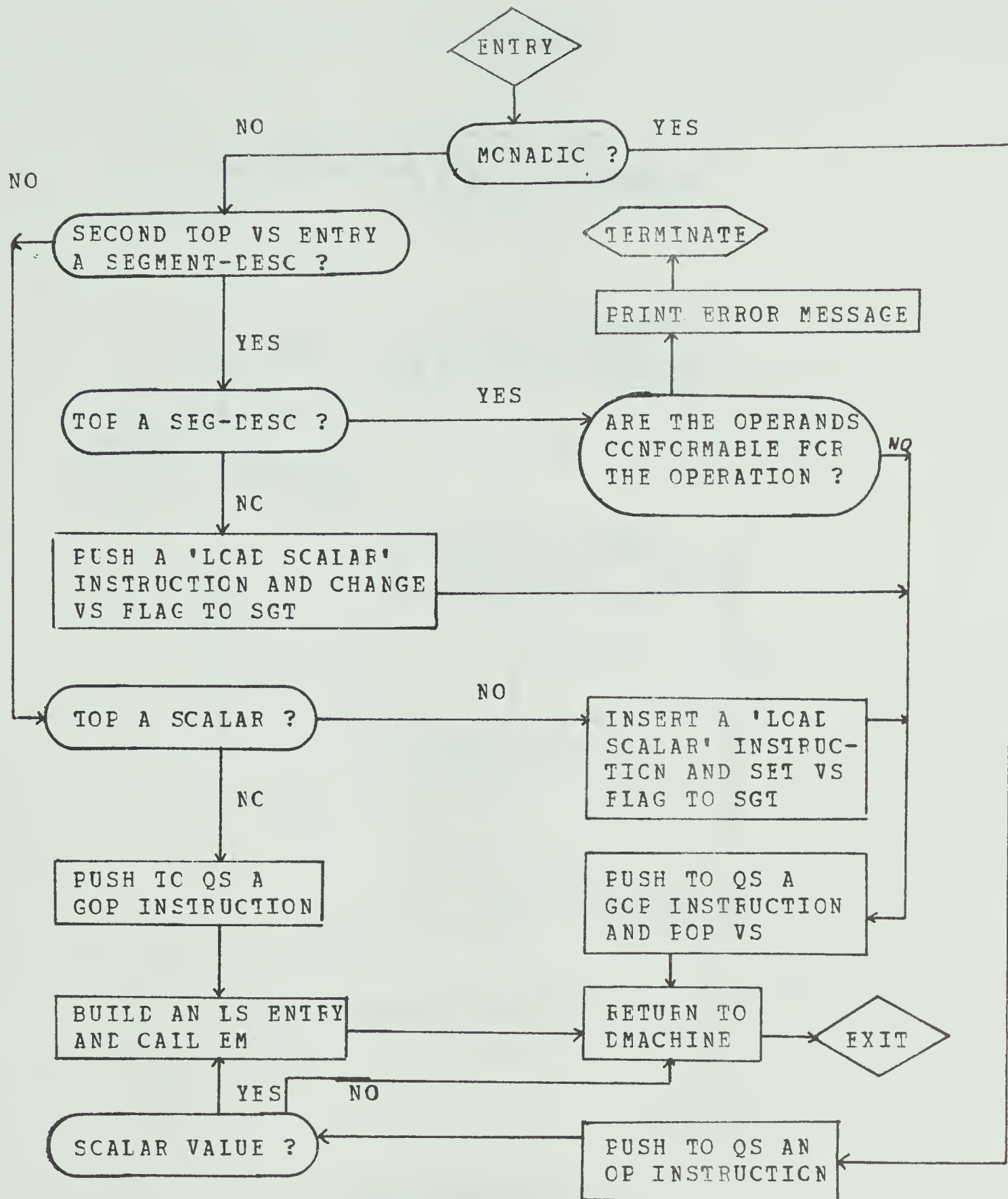




DMACHINE FLOW-CHART

Figure 8.3



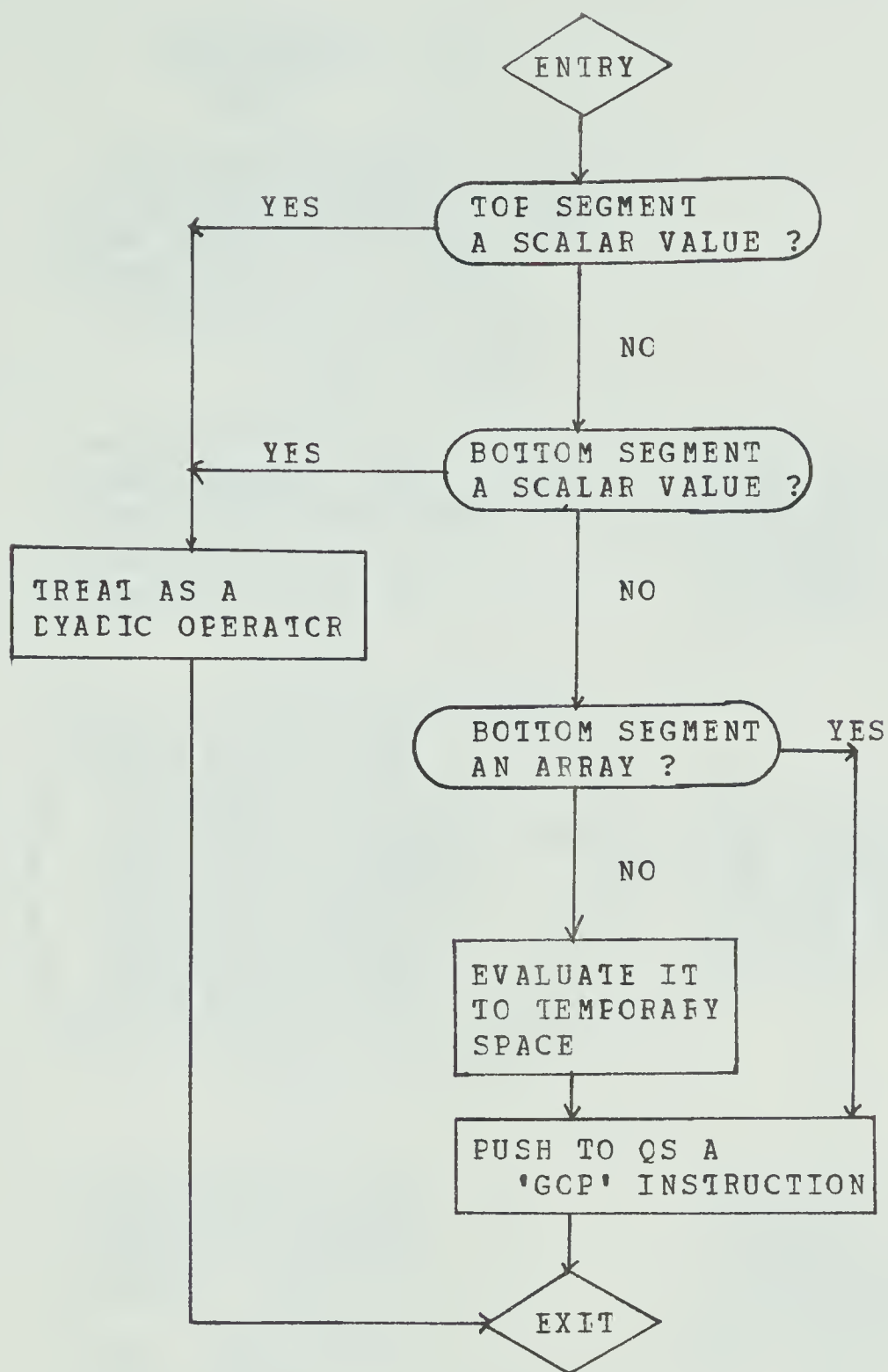


MDADIC FLOW-CHART

Figure 8.4



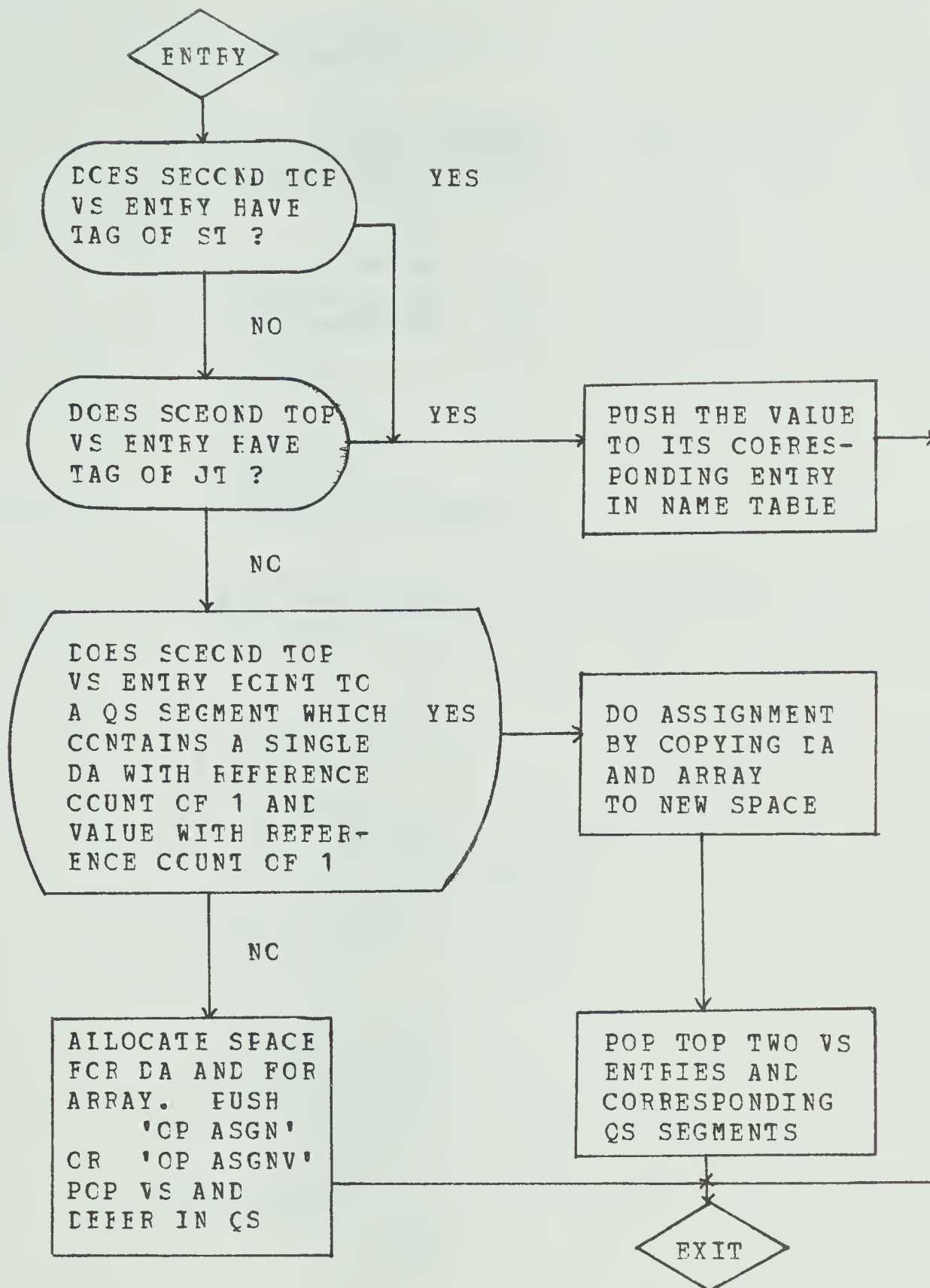




GDF FLOW-CHART

Figure 8.5

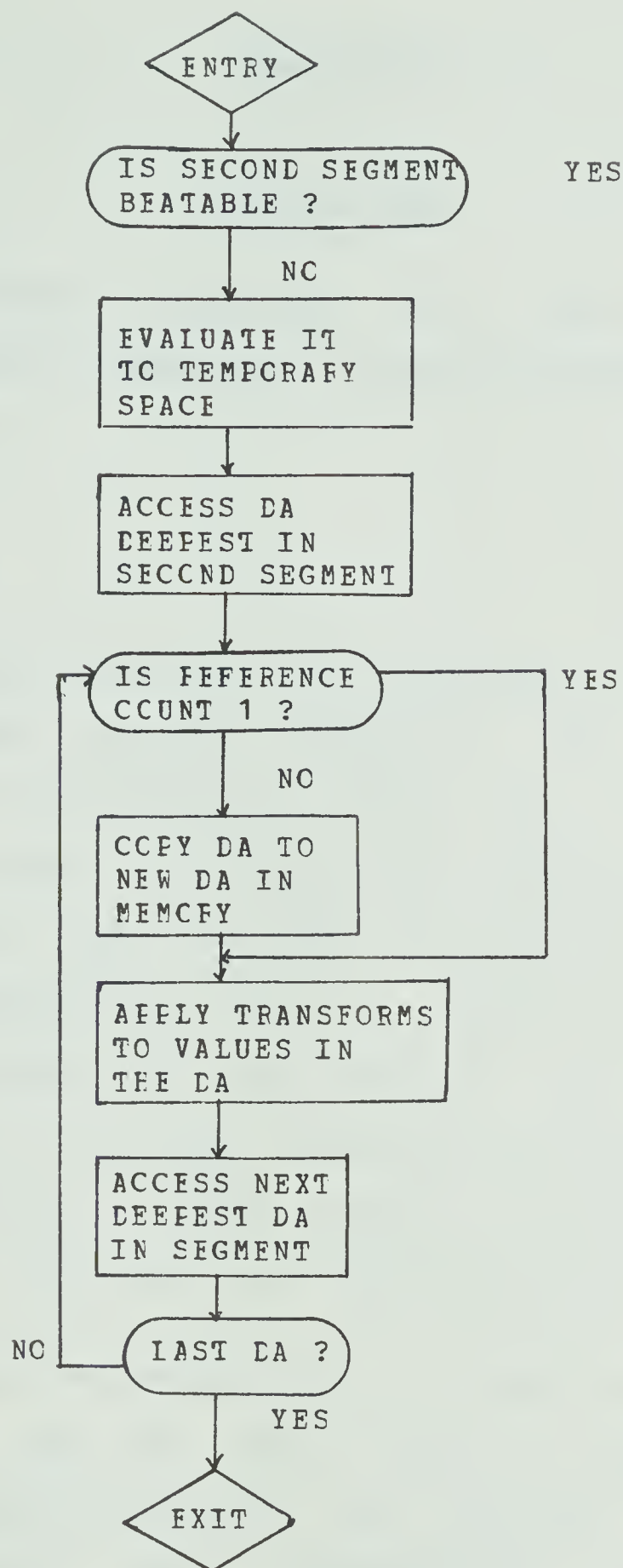




ASGN FLOW-CHART

Figure 8.6





SELECT FLOW-CHART

Figure 8.7



## Conclusions

### Chapter Nine

This chapter summarizes the APLM and outlines extensions and refinements of the research and simulation already done.

#### 9.1 Summary.

Abrams has shown that APL expressions containing only selection and reduction operations, and inner and outer products, may be reduced to a simpler set of operations on the same operands. He developed the theoretical basis for a machine which could do those reductions by a well-defined set of transformations, which are applied to the storage access functions of the operands. He further stated conditions for which those transformations could be applied. He also outlined a stack oriented machine which could theoretically implement those transformations.

The author has implemented, by simulation, a sub-set of the machine, the DM, which analyzes monadic and dyadic scalar arithmetic operators and selection operators and outer products. The implementation of this sub-set illustrates the theoretical basis for the machine and





interfaces directly with the execution sub-set of the machine, EM, written by Appleyard. Another sub-set has been outlined, the CM, which compiles programs into program segments.

The system outlined by Abrams indicated that outer products could be included in the class of operations, which could be transformed, only if both the right and left-hand operands are array values. Analysis of the outer product operation showed that only the right-hand operand should be an array. The left-hand operand may be an array-valued expression. The right-hand operand has all its elements accessed for each access of an element of the left-hand operand. It is therefore more efficient to evaluate the right-hand operand to temporary space before performing the outer product operation. The left-hand operand has each of its elements accessed only once, therefore it is more efficient not to evaluate it to temporary space. This extension allows outer product operations to be deferred, and beaten, after a maximum of one immediate evaluation compared with a maximum of two, as indicated by Abrams.

The simulation includes the implementation of the monadic and dyadic scalar arithmetic operators, outer products, and the operations take, drop, reversal and transpose. The implementation of these operators is aided



by the processes of drag-along and beating, and therefore is included. Operations such as inner products are similar to previously mentioned operations, and are therefore not included. The operators which must be executed immediately do not facilitate drag-along and beating and are not included because they would differ little from present implementations. The remaining operators are not simulated because they do not further illustrate the processes of drag-along and beating, however their descriptions and methods of implementation are contained in the thesis.

## 9.2 Extensions and Refinements.

As in present implementations, the addition of bulk I/O would increase accessibility to the problem-solver. The addition of secondary storage would greatly increase the capacity of the system for data and programs alike. Several logical units of the system are often used for routine tasks, such as the NT search unit. The indexing unit, which determines the location of an element in an array using the storage access function, is used frequently and therefore should be as efficient as possible. MAINCYCL should also be very efficient. Therefore, research should be carried out on implementing these units by micro-program, thereby reducing the overhead at execution time in terms of central processor requirements.



Future investigation would be well spent on examination of the processes for the beating of general subscripted expressions. Although subscripting was not implemented in this simulation, it was found that subscripted expressions can be beaten by beating the components of the subscripted expression. Abrams stated that subscripted expressions could be beaten only if the subscriptee itself is beatable. Investigation by Appleyard and the author have shown that a subscripted expression may be beaten if the individual components of the subscriptee expression are beatable. This condition applies to the selection operators take, drop, and reverse, but only to transpose under certain circumstances, which are further described in Appleyard's thesis. For example,

$$3 \ 4 \ 5 \ \uparrow \ M[A;B;C] \leftrightarrow M[(3 \uparrow A);(4 \uparrow B);(5 \uparrow C)]$$

The take operation could therefore be accomplished by beating the components of the subscriptee by the elements of the take operand, rather than the calling of the EM to evaluate the subscriptee to an intermediate result. This extension further minimizes unnecessary evaluations to execute programs written in API.



## APPENDIX A.

MCNITOR AND INITIALIZERS.

MCNITOR LISTING.

SPECARR LISTING.

SPECIS LISTING.

DMACCCDE LISTING.





```

        TITLE 'THE MCNITOR'
        ENTRY MONITOR
MCNITOR  LS      0H
        B      START(,15)
*
*  INITIALIZATION OF AFIM CONSTANTS
        LS      0D
MONISAVE DC      9C'PROGSAVE'  PROGSAVE
        DC      A(CCRELCW-4)  MEMADDR
        EC      1F'0'        ARRAVAIL
        DC      A(*-4,*-8)
        DC      1F'0'        LAAVAIL
        DC      A(*-4,*-8)
        DC      A(CCORELOW)    BOTTPOOL
        DC      A(CCOREHIGH)   TOPPOOL
        DC      A(DS11)        ISMARK
        DC      1F'0'          QSCNTRL
        DC      1D'0'          DBL
*
        DC      C'SAVEARR '
        EC      A(DS12)        ISEND
        DC      A(DS11)        ISTOP
        DC      A(DS11)        ISSCAN
        DC      A(DS11-ISWIDTH) ISEASE
*
        DC      A(DS22)        LSEND
        EC      A(DS21)        LSTOP
        DC      A(DS21)        LSSCAN
        DC      A(DS21-LSWIDTH) LSBASE
*
        DC      A(DS32)        QSEND
        DC      A(DS31)        QSTCP
        DC      A(DS31)        QSSCAN
        DC      A(DS31)        QSBASE
*
        DC      A(DS42)        VSEND
        DC      A(DS41)        VSTOP
        DC      A(DS41)        VSSCAN
        DC      A(DS41-VSWIDTH) VSBASE
*
        EC      A(DS212)        NTEND
        DC      A(DS211)        NTOP
        DC      A(DS211)        NISCAN
        DC      A(DS211)        NTEASE
        DC      1F'0'
*
        DC      2F'0'          VSTAGERR AND VSTYPERR
        EC      1X'CO'        NEWIT AND STEPTOG
*

```



```

          DS      0D                      MEMORY
          DC      A ((COREHIGH-CORELOW) - 1) MEMORY 1K LONG, MEMEND
CORELOW   DS      0H
          DS      511F
COREHIGH  DS      0H
START     EQU     *-MCNITOR
          STM      14, 12, 12 (13)
          LR       12, 15
          USING    MCNITOR, 12
          ST       13, MCNISAVE+4
          LR       10, 13
          LA       13, MCNISAVE
          ST       13, 8 (, 10)
          USING    APLMCCNS, 13
          CALLO    DMACCCDE
          CALLO    SPECIS
          CALLO    SPECIS
          CALLO    SPECVS
          CALLO    SPECARR
          CALLO    SPECDA
          CALLO    SPECNT
          CALLO    EMACCCDE
          ST       12, MCNIBASE
          L        15, =V (MAINCYCL)
          L        14, BCTTPOOL
          BALR     10, 15
          L        12, 8 (, 10)
          B        START1
MONIBASE  DS      1F
START1    L        13, MCNISAVE+4
          LM       14, 12, 12 (13)
          BR       14
          DROP     12, 13
          LTORG
          DC       5F'15, 291'
DS11      DS      0F
          DC       (5*ISWIDTH) C'ISREG'
DS12      DS      1F
DS21      DS      0F
          DC       (5*LSWIDTH) C'LSREG'
DS22      DS      1F
DS211     DS      0F
          DC       (20*NTWIDTH) C'NTREG'
DS212     DS      1F
DS31      DS      0F
          DC       (15*QSWIDTH7) C'QSREG'
DS32      DS      1F
DS41      DS      0F
          DC       (10*VSWIDTH) C'VSREG'
DS42      DS      1F

```



	ENTRY SPECARR	
SPECARR	DS 0D	
	DMENT SPARRSVE, SPECARR	
SPSTART	EAL 10, SPREAD	READ CARD, END OR STAFT
	CIC SPTEMP(4), =C'END '	CHECK IF END CARD
	BF SPFINISH	IF SO, FINISHED
	CLC SPTEMP(4), =C'STAR'	CHECK IF START CARD
	BNE SPERROR	IF NOT, AN ERRCR
*	READ IN INFORMATION ABOUT	THE DA
	BAL 10, SPREAD	READ IN THE DA LENGTH
	SR 2, 2	
	L 3, SPTEMP+4	LOAD DA LENGTH
	BAL 10, SPHEX	CCNVERT TO BINARY REG 1
	LR 4, 1	FOR LATER USE
	L 9, TCFPOOL	ACCESS TOPPOOL
	SR 9, 1	ALLOCATE DA SPACE
	ST 9, TCFPOOL	RESTORE NEW TOPPOOL VALUE
	USING DESCRIPT, 9	ACCESS NEW DA
*	BUILD NT ENTRY	
	BAL 10, SPREAD	READ IN INX VALUE
	SR 2, 2	
	L 3, SPTEMP+4	LOAD INX VALUE
	BAL 10, SPHEX	CCNVERT TO BINARY
	L 8, NTTOP	ACCESS TOP OF NT
	LA 8, NTWIDTH(, 8)	SET UP FOR NEW ENTRY
	ST 8, NTTOP	RESTORE NEW NTTOP VALUE
	USING NTREG, 8	ACCESS NEW NT ENTRY
	ST 2, NTVALUE	ZERO WORD 2
	STH 1, NTINX	SET INX VALUE
	MVI NTTAG, TAGDT	SET TAG FOR DA POINTER
	LR 7, 9	
	S 7, MEMADDR	MAKE ABSOLUTE
	ST 7, NTVALUE+4	SET DA POINTER
	MVI NTTYPE, B'00000000'	ZERO TYPE FLAG
	DROP 8	FINISHED WITH NT ENTRY
*	BUILD DESCRIPTOR ARRAY	
	ST 4, DALEN	SET UP LENGTH
	ST 2, DAABASE	ZERO ABASE
	MVI DATYPE, TYPINT	SET TYPE TO INTEGER
	MVC DAFILI(4), =X'00000001'	SET FILLER AND RC
	L 7, BCTTFCOL	GET VBASE VALUE
	S 7, MEMADDR	MAKE ABSOLUTE
	ST 7, DAVBASE	SET VBASE VALUE
	DROP 9	FINISHED WITH DSECT
	LA 9, 16(, 9)	SKIP UP TO RANK ETC.
	S 4, =F'16'	AMOUNT LEFT TO READ IN
	SRL 4, 2	IN WORDS
	BAL 8, SPREADIN	READ IN REST OF DA
*	READ IN THE ARRAY OF DATA	



BAL	10,SPREAD	READ IN LENGTH OF ARRAY
SR	2,2	
L	3,SPTEMP+4	LOAD LENGTH
BAL	10,SPHEX	CONVERT TO BINARY
LR	4,1	FOR LATER USE
L	9,BCTTECCL	START OF ARRAY
BAL	8,SPREADIN	READ IN ARRAY
ST	9,BCTTECCL	RESTORE BOTTPPOOL
B	SPSTART	START AGAIN
* LOCAL READ ROUTINE		
SPREAD	READ 1,SPTEMP	READ IN CARD
	WRITE 6,SPTEMP,20	WRITE OUT THE CARD
	BR 10	
*LOCAL CCNVERT ROUTINE		
SPHEX	CALL HEXCCNV	CCNVERT REG 2&3 INTO REG 1
	BR 10	RETURN
* READ INTO MEMORY		
SPREADIN	EAL 10,SPREAD	READ IN CARD
	LM 2,3,SPTEMP	LOAD IT
	BAL 10,SPHEX	CONVERT IT
	ST 1,0(,9)	STORE IT IN MEMORY
	LA 9,4(,9)	INCREMENT POINTER
	BCT 4,SPREADIN	RETURN IF NOT FINISHED
	BR 8	RETURN
* ERROR HAS BEEN FOUND		
SPERROR	WRITE 6,'LENGTH ERROR IN SPECARR'	
	DC 1H'00'	BCMB
* INITIALIZATION IS FINISHED		
SPFINISH	DMEX SPARRSVE	
SPTEMP	CS 20F	
	DROP 8,9,11,12	
	LTORG	
	DS 10F	





	TITLE	'SPECIFY LOCATION STACK'	
	ENTRY	SPECIS	
SPECIS	DS	0D	
	DMENT	SPECLSVE,SPECIS	
	READ	5,LSINPUT,4	READ IN INX VALUE
	SF	2,2	ZERO REG 2
	L	3,LSINPUT	
	CALL	HEXCONV	REG 1 HAS BINARY INX
	CALL	NTSEARCH	REG 1 ADDRESS NT ENTRY
	L	6,LISTOP	
	LA	6,LSWIDTH(0,6)	SET UP FOR NEW LS ENTRY
	ST	6,LISTOP	
	USING	LSREG,6	ACCESS NEW LS ENTRY
	L	5,8(,1)	GET POINTER TO SEG DESC
	A	5,MEMADDF	MAKE ABSOLUTE
	USING	SEGDREG,5	ACCESS SEG DESC
	L	4,SDORG	GET LENGTH FIELD
	S	4,=F'8'	CUT OFF HEADER
	ST	4,LSDE	SET UP LENGTH FIELD
	L	4,SDFISR	ACCESS BASE OF PROG SEG
	N	4,=X'00FFFFFF'	BLANK BYTE ONE
	A	4,MEMADDR	MAKE ABSOLUTE
	LA	4,8(,4)	SET ORG UP 8 BYTES
	ST	4,LSCRG	TO SKIP HEADER
	SR	4,4	
	ST	4,LSREL	SET REL TO 0
	ST	4,LSQF	SET QP TO ZERO
	MVI	LSDE,B'00000000'	DE BIT TO 1 CTHERS TO 0
	CMEX	SPECLSVE	
	DROP	5,6,11,12	
LSINPUT	DS	20F	
	DS	10F	



```

      TITLE 'DMACCCCE SECTION'
DMACCODE CSECT
      DMENT DMACSAVE,DMACCODE
DMACSTRT BAL      7,READ          READ IN START OR END CARD
      CLC      INPUT(4),=C'END '  CHECK FOR END CARD OF SEG
      BE      FINISH          IF SO, FINISHED INITIALIZING
      CLC      INPUT(4),=C'STAR'  CHECK IF START OF NEW SEG
      BNE      ERRCR          IF NOT, THERE AN ERROR
      LA      7,READ1
READ      READ      1,INPUT          GENERAL READ FOR CSECT
WRITER    WRITE    6,INPUT,20
      BR      7
HEX        CLC      INPUT(4),=C'END '
      BE      ERRCR          IF END CARD, LENGTH ERROR
      CLC      INPUT(4),=C'STAR'  IF START CARD, EPROR
      BE      ERRCR
      CALL    HEXCONV          CONVERT HEX IN REG 2&3
      BR      7
READ1      L        3,INPUT
      BAL      7,HEX          1 HAS INX VALUE CF SEG
      LR      9,1            STORE FOR INSERTION IN NT
      BAL      7,READ          READ FISR, FPARS, FLCIS
      LM      2,4,INPUT
      BAL      7,HEX          REG 1 HAS FISR & FPARS
      ST      1,DMACPARS      STORE FOR LATER USE
      SR      2,2
      LR      3,4
      BAL      7,HEX          REG 1 HAS FLCIS
      STH      1,DMACPARS+4
*          DMACPARS CONTAINS FISR, FPARS, FLCIS IN 3 HAIF-WORDS
      AH      1,DMACPARS+2    ADD FPARS
      AH      1,DMACPARS      ADD FISR
      STH      1,DMACPARS+6    STORE FOR LATER USE
      SIL      1,1            TO GET # OF BYTES
      LA      1,20(,1)        ADD 20 FOR PREFIX ETC.
*          REG 1 HAS LENGTH CF SEGMENT DESCRIPTOR
      L        5,TCFPOOL      GET TOP OF AVAILABLE MEM
      SR      5,1            ALLOCATE SPACE FOR DESC
      N        5,=X'FFFFFFFC' GO TO LOWER FULL WORD
      ST      5,TCFPCCL      RESTORE TOPPOOL VALUE
      ST      1,0(,5)        WORD 1 OF PREFIX, LENGTH
      MVC      4(4,5),=X'00010000' SET RC=1 AND FILLER=0
      STM      1,15,DEBUG1
*          BUILD THE NT ENTRY
      L        10,NTICP
      LA      10,NTWIDTH(,10) INCREMENT NEW NT ENTRY
      ST      10,NTTOP        RESTORE NEW NTICP VALUE
      USING   NTREG,10        ACCESS NEW NT ENTRY
      SR      3,3

```



IR	4,3	ZERO OUT WORDS 1 AND 2
S	5, MEMADDE	MAKE ADDRESS RELOCATABLE
STM	3,5, NTINX	PUT TOPPOOL INTO VALUE
A	5, MEMADDE	MAKE ADDRESS ABSOLUTE
STH	9, NTINX	SET UP INX VALUE
MVI	NTIAG, TAGSGT	SET TAG TO SEGMENT
*	BUILD WORD 1 OF DESCRIPTOR	FISR FVBASE
*	BUILD WORD 2 OF DESCRIPTOR	FIORG FLEN+8
*	BUILD WORD 3 OF DESCRIPTOR	FPARS FLCLS
BAL	7, READ	READ FIORG AND FLEN 2A4
IM	2,3, INFUT	
BAL	7, HEX	REG 1 HAS FIORG AND FLEN
LA	1,8 (, 1)	REG 1 HAS FIORG, FLEN+8
LR	4, 1	SAVE FOR LATER USE
LR	8, 5	REG 8 HAS TOPPOOL
USING	SEGDREG, 8	START SEGMENT DESCRIPTOR
ST	1, SDCRG	NOW IN WORD 2 OF SEG DESC
L	7, BCTIPOOL	ACCESS BOTTOM OF POOL
IR	6, 7	
S	7, MEMADDE	
ST	7, SDFISR	SET FVBASE
MVC	SDFISR (1), DMACPARS+1	SET FISR
MVC	SDFPARS (4), DMACPARS+2	SET FPARS AND FLCLS
LH	9, DMACPARS+6	GET FISR+FPARS+FLCLS
STM	1, 15, DEBUG2	
LTR	9, 9	
BZ	BLDHEADR	IF THERE NO PARAMETERS
LA	5, 20 (, 5)	SET REG 5 TO LIST PARAMS
*	LIST PARAMETERS IN SEGMENT DESCRIPTOR	
LISTPARS	BAL 7, READ	
I	3, INFUT	
BAL	7, HEX	REG 1 HAS INX VALUES
STH	1, 0 (, 5)	STORE BYTES 3 AND 4
LA	5, 2 (, 5)	INCREMENT BY 2 BYTES
BCT	9, LISTPARS	DECREMENT # OF PARAMETERS
*	BUILD HEADER OF PROGRAM SEGMENT	
BLDHEADR	N 4, =X'00FFFFFF'	ISOLATE FLEN+8
ST	4, 0 (, 6)	SET LENGTH AND PREFIX
S	4, =F'8'	NOW HAS LENGTH OF CODE
MVC	4 (4, 6), =X'00010000'	SET RC=1 AND FILLER=0
LA	6, 8 (, 6)	SET PAST WORD 2 OF PREFIX
STM	1, 15, DEBUG3	
*	ACCESS DM INSTRUCTION	
LICCODE	EAL 7, READ	READ IN INSTRUCTION
LM	2, 3, INFUT	
BAL	7, HEX	REG 1 HAS FIRST 4 BYTES
ST	1, 0 (, 6)	STORE WORD OF PROGRAM
LA	6, 4 (, 6)	INCREMENT INDEX BY 4
S	4, =F'4'	DECREMENT LENGTH BY 4
BP	IDCCDE	RETURN IF MORE DMCCDE



	ST	6,BCTTFCOL	UPDATE BOTTPPOOL
	STM	1,15,DEBUG5	
	B	DMACSTRT	CHECK IF PROPER END
ERROR	MVC	INPUT(20),ERROR1	WRITE OUT LENGTH ERROR
	DC	1H'CO'	
FINISH	DMEX	DMACSAVE	
INFUT	DS	20F	
DMACPARS	DS	3F	
DEBUG1	DS	20F	
DEBUG2	DS	20F	
DEBUG3	DS	20F	
DEBUG5	DS	20F	
ERROR1	LC	C' SEGMENT LEN ERROR	'
	DROP	8,10,11,12	
	LTORG		
	DS	8F	





## APPENDIX E.

## MAINCYCL LISTING.



```

ENTRY MAINCYCL
MAINCYCL DS      OH
          RECURENT
          L        ISFNT, LSTOP
          USING LSREG, ISFNT
*STEP (1)
CYCLSTRT CIC      ISREL+1(3), LSLEN      REL & LEN SHOULD BE +
*                                           SO LOGICAL COMPARISON WORKS
          BI        CYCIDCRE
*STEP (2)
          TM        ISTRT, LSTRTON      TEST IF LS STRUCT IS ON
          BNO       CYCIISTS           BRANCH TO CYCLE IS TEST
          L         14, EOTTECOL
          LR        12, 14
          L         15, =V(POPIS)
          BALR      10, 15
          NI        LSTRT, LSTRTCFF     TURN OFF LS STRUCT
          B         CYCLRELO
*STEP (3)
CYCLISTS TM        ISIS, ISISCN        DOES LS HAVE CCNTROL?
          EZ        CYCLPCPI
*STEP (4)
CYCLRELO SR        TEMF, TEMF          ZERO LSREL
          ST        TEMF, ISREL
          STM       0, 15, DEBUG100
          CALLREC   STEPIS             INCREMENT INDEX GENERATOR
          NI        NEWIT, NEWITCFF    AFTER INCREMENT, OLD ITERAT
          TM        STEFTOG, STPTIGON  HAVE RUN THROUGH INDICES
          BO        CYCISTRT
*STEP (5)
          L         ISFNT, ISTOP
          USING     ISREG, ISFNT
          TM        ISTRT, ISTRTON     TEST IF IS STRUCT IS ON
          BNO       CYCLPCPI           BRANCH TO POP LS IF NOT
          OI        LSTRT, LSTRTON
          B         CYCISTRT
*STEP (6)
          STM       0, 15, DEBUG101
CYCLPCPI CALLREC   LFCF
          B         CYCISTRT
          USING     QSREG1, QSFNT
*STEP (7)
CYCLDORE TM        LSDE, EMAC          EM OR DM IN CONTROL
          BO        EMACHINE
          L         15, =V(DMACHINE)
          BALR      14, 15
          E         CYCISTRT
*STEP (8)
EMACHINE L         QSFNT, QSEBASE      QSBASE

```



```

      A      QSPNT,LSREL      QSBASE+LSORG
      A      QSPNT,LSCRG      QSBASE+LSORG+LSREL
*STEP (9)
      SR      TEMP1,TEMP1
      IC      TEMP1,QSOP1      ACCESS INSTRUCTION INFO
*                                     I.E. ATTRIBUTES AND ADDRESS
*                                     OF ROUTINE
      LR      TEMP,TEMP1
      N      TEMP,=X'0000003F'
      SLL     TEMP,2          DISPLACEMENT IN QSOPTABLE
      L      TEMP,QSOPTABL(TEMP) ADDRESS OF ROUTINE
      LTR     TEMP,TEMP      IS INSTRUCTION MARKED
      BM      CYCLSCAN
*STEP (10)
      N      TEMP1,=X'000000C0' ZERO BITS 0-23 AND 26-31
      SRL     TEMP1,4        NO. OF BYTES IN INSTRUCT
      A      TEMP1,ISFEL     UPDATE RELATIVE IS POINTER
      ST      TEMP1,LSREL
      TM      ISTFT,ISTRTON
      BNO     CYCIREP
      SIL     TEMP,1
      LTR     TEMP,TEMP
      EM      CYCLSTRT
      SRL     TEMP,1
      B      CYCIREP
*STEP (11)
CYCLSCAN TM      ISTFT,ISTRTON
      BNO     CYCIREP
      DC      1H'0'
CYCLREP  L      14,BOTTPCOL      RECURSIVE CALL
      LR      12,14
      STM     0,15,DEBUG102
      BALR    10,CAI1ALDR
      L      ISFNT,LSTOP
      B      CYCLSTRT
      DROP    QSPNT,ISFNT,ISFNT
      ITORG
DEBUG100 DC      16A(291)
DEEUG101 DC      16A(291)
DEEUG102 DC      16A(291)

```



## APPENDIX C.

## DMACHINE LISTING.





```

        TITLE 'MAIN BODY OF THE D-MACHINE'
        ENTRY DMACHINE
DMACHINE DS      0C
*        REG 8 HAS OFCCDE 3 FOR LEAVE, AND 4 FOR RETURN
        DMENT2 DMACHSVE,DMACHINE
        L        10,ISTOP
        USING ISREG,10                ACCESS TOP OF IS
*        CHECK IF EM WAS DOING AN INTERMEDIATE EVALUATION
        CLC      TEMPRET(4),=X'00000000' CHECK IF ZERC
        BE        DMCHKTAG            IF SO, CONTINUE
        L        15,TEMPRET          OTHERWISE, BRANCH
        BR        15                TO ADDRESS IN TEMPRET
*        CHECK IF VS TAG IS FT, FDT, OR JT
DMCHKTAG L        10,VSTCF
        USING VSREG,10                ACCESS TOP ENTRY OF VS
        MVC      DMTEMP(1),VSTAG      STCRE VSTAG FOR LATER USE
        CLI      DMTEMP,TAGFDT        CHECK IF TAG JT,FT,FDT
        BH        DMSTART            IF NOT, START NEW INSTRUC
*        PUSH QS ENTRY OF OP ARRAYPOINT MASK
        MVI      VSTAG,TAGSGT        SET TAG TO SGT
        L        9,QSINSFT           ACCESS TOP ENTRY OF QS
        ST        9,QSTOP            UPDATE QSTOP
        LR        8,9                FOR LATER USE
        LA        8,QSWIDTH5(,8)     SPACE FOR NEW QS ENTRY
        ST        8,QSINSRT          NEW QSINSRT VALUE
        USING QSREG5,9              THIS WILL BE TYPE QSREG5
        SR        2,2                FOR ZEROING
        ST        2,QSOP5            ZERO OUT
        ST        2,QSADDR5          THE QS
        ST        2,QSMASK5          INSTRUCTION
        CLI      DMTEMP,TAGDT        CHECK IF JT, DT, OR FDT
        BH        DMFDT              BRANCH FOR FDT
        BI        DMJT              BRANCH FOR JT
        MVI      QSOP5,OPIA          DT, SET OPCODE TO IA
        B        DMMASK              BUILD THE MASK
DMFDT    MVI      QSOP5,CPIFA        FDT, SET OPCODE TO IFA
        B        DMMASK              BUILD THE MASK ETC.
DMJT    MVI      QSOP5,OPIJ          JT, SET OPCODE TO IJ
        B        DMSTART            ACCESS THE INSTRUCTION
*        OP PART OF ENTRY HAS BEEN CCNSTRUCTED
DMMASK  L        7,VSVALUE+4        GET DA POINTER
        ST        7,QSADDR5          LOAD IT INTO ARRAYPOINTER
*        ARRAYPOINTER HAS BEEN CCNSTRUCTED
        SR        2,2
        L        3,=X'FFFFFFFF'      FILL REG 3 WITH CNES
        L        4,16(,7)           OBTAIN RANK
        SLDA     2,0(4)              FILL REG 2 WITH RANK CNES
        ST        2,QSMASK5          SET MASK
*        MASK HAS BEEN CCNSTRUCTED

```



*	NOW MUST PUSH AN IS ENTRY	FOR EACH DIMENSION
	L 6,16(,7)	RANK AS THE INDEX
	LA 7,20(,7)	UP TO NEXT DIMENSION
	L 5,ISTOP	ACCESS TOP OF IS
	LA 5,ISWIDTH(,5)	ALLOCATE NEW ENTRY
	ST 5,VSVALUE	SET IS POINTER
	USING ISREG,5	ACCESS NEW IS ENTRY
	MVI ISDIR,B'00010000'	SET FLAGS
	SR 2,2	ZERO REG 2
DMDIME	ST 2,ISCTR	ZERO CCOUNTER
	MVC ISMAX(3),1(7)	SET LENGTH FIELD
	BCT 6,DMDIME1	BRANCH IF NOT LAST DIMEN
	ST 5,ISTCF	SET NEW ISTOP VALUE
	B DMSTART	THEN FINISHED
DMDIME1	LA 5,ISWIDTH(,5)	UP TO NEXT IS ENTRY
	LA 7,8(,7)	UP TO NEXT DIMENSION
	MVI ISDIR,B'00000000'	SET FLAGS
	B DMDIME	RETURN FOR NEXT DIMENSION
DMSTART	L 10,ISTOP	
	USING LSREG,10	ACCESS TOP OF LS
	L 9,ISCRG	
	A 9,ISREI	GET ADDRESS OF INSTRUCT
	LH 8,0(,9)	LOAD 2 BYTES CF CODE IN 8
	LR 5,8	ALSO IN REG 5
	LR 6,8	ALSO IN REG 6
	LR 7,8	ALSO IN REG 7
	N 8,=X'0000003F'	ISOLATE OPCODE IN REG 8
	SIL 7,16	
	SRL 7,28	ISOLATE LENGTH IN REG 7
	C 8,=F'39'	
	BH DMLDS	BRANCH IF LDS CR LDJ INS
	C 7,=F'4'	
	BE DMLLEN4	INSTRUCTION LENGTH IS 4
	BH DMLLEN6	INSTRUCTION LENGTH 6, 10
DMLLEN2	SIL 6,20	
	SRL 6,26	ISOLATE K OR APL OF IN 6
	E DMERANCH	
DMLLEN4	LH 5,2(,9)	ACCESS BYTES 3&4
	N 5,=X'0000FFFF'	REG 5 HAS INX OR COORD
	B DMLLEN2	REG 6 HAS 24 BIT ADDRESS
DMLLEN6	MVC DMTEMP(8),2(9)	ACCESS BYTES 3-10
	LM 3,4,DMTEMP	REG 3 CONTAINS ADDRESS
	B DMERANCH	
DMLDS	SIL 6,20	
	SRL 6,28	ISOLATE TYPE IN REG 6
	C 7,=F'2'	CHECK LENGTH AGAIN
	BNE DMLLEN6	REG 3 CONTAIN AN INTEGER
*		CHARACTER, OR REAL VALUE
*		REG 3&4 CONTAIN LREAL
*		REG 3&4 CONTAIN J-VECTOR



	SIL	5,24	
	SRL	5,31	BINARY VALUE IN REG 5
DMERANCH	L	9,ISREL	
	AR	9,7	
	ST	9,ISREL	UPDATE REL
	LR	9,8	DISPLACEMENT INTO DMOPTBL
	SIL	9,2	IN BYTES
	L	15,DMOPTBL-4(9)	GET ROUTINE ADDRESS
	BAIR	14,15	BRANCH AND LINK TO ROUTINE
	DMEX	DMACHSVE	
DMTEMP	DS	20F	
DMOPTBL	DC	V(ASGN)	
	DC	V(ASGN)	
	DC	V(LEAVE)	
	DC	V(LEAVE)	
	DS	F	
	DS	F	
	DS	F	
	DC	V(SELECT)	
	DC	V(SELECT)	
	DC	V(SELECT)	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DC	V(LIS)	
	DC	V(LDCCN)	
	DC	V(JMP)	
	DC	V(JMP)	
	DC	V(JMP)	
	DC	V(SELECT)	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DS	F	
	DC	V(MIADIC)	
	DC	V(MIADIC)	
	DC	V(GDF)	
	DC	V(LNF)	
	DC	V(LNF)	
	DS	F	
	DC	V(LCAD)	



DC        V (LCAD)  
EC        V (LCAD)  
LTORG  
DS        10F





## APPENDIX D.

## ICAD LISTING.



```

TITLE 'ICAD SECTION'
ENTRY ICAD
LOAD    LS      0D
        DMENT LOADSAVE,LOAD
*       REG 8 HAS CFCODE 24=LDN, 27=LDSEG
*               28=LDS, 29=LDJ
*       REG 3 HAS SINGLE WORD OPERAND
*       REG 3&4 HAVE DOUBIE WORD OPERAND
*       REG 5 HAS BINARY VALUE
*       REG 6 HAS TYPE FOR LDS INSTRUCTION
*       THIS CSECT FCB LDS, LDSEG, LDJ AND LDN
        L      10,VSTOP
        LA     10,VSWIDTH(,10)      SET UP FOR NEW INTRY
        ST     10,VSTOP             STORE NEW VSTOP VALUE
        USING VSREG,10             ACCESS NEW ENTRY
*       PUSH THE DOUBIE WORD OPERAND TO THE VALUE FIELD
*       CTHER CASES HANDLED SEPARATLY
        SIM    3,4,VSVALUE          STORE DOUBIE WORD OPERAND
        C      6,=F'28'             CHECK FOR LDS=28
        BL     LONSEG               EITHER LDN OR LDSEG
        BE     LCS                  IF EQUAL, THEN LDS
*       OTHERWISE HAVE ISCLATED IDJ INSTRUCTION
        MVI    VSTAG,TAGJT          SET TAG TO JT
        B      LOFINISH             THEN FINISHED
*       HAVE ISOLATED LDS INSTRUCTION
LOS     MVI    VSTAG,TAGST          SET TAG FOR SCALAR
        ST     6,LCTEMP             STORE TYPE FOR LATER USE
        MVC    VSTYPE(1),LCTEMP+3  SET TYPE
        TM     LCTEMP+3,B'10000000' TEST FOR BOOIFAN
        BZ     LONCTBOC             IF NOT, NOT BOOIFAN
*       HAVE ISOLATED BOOIFAN SCALAR
        ST     5,VSVALUE+4          STORE BINARY VALUE IN VS
        B      LCFINISH             THEN FINISHED
*       HAVE EITHER LDN OR LDSEG
LONSEG  C      6,=F'27'             CHECK FOR OPCODE LDSEG
        BE     LOLDSEG              IF SO THEN ERANCH
        MVI    VSTAG,TAGNPT         OTHERWISE TAG SET IC NPT
        ST     3,VSVALUE+4          STORE INX VALUE
        B      LOFINISH             THEN FINISHED
*       HAVE EITHER IREAL OR A 4 BYTE OPERAND
LONOTBOO TM     LCTEMP+3,B'00001000' TEST FOR IREAL
        BO     LCFINISH             IF SO THEN FINISHED
        ST     3,VSVALUE+4          STORE 4 BYTE VALUE
        B      LOFINISH
*       HAVE ISCLATED LDSEG
LOLDSEG ST     3,VSVALUE+4          SET VALUE TO SD POINTER
        MVI    VSTAG,TAGSGT         SET TAG TO SEG DESCRIPTOR
LOFINISH SIM    0,15,LCTEMP         DUMP REGISTERS
        DMEX   LOALSAVE

```



LOTEMP    DS    20F  
          DROP   10,11,12  
          LTORG  
          DS    10F



## APPENDIX E.

CONTROL AND STORAGE ACCESS.

INF LISTING.

JMP LISTING.





ENTRY LNF		
DS 0D		
LNF	REG 5 HAS INX VALUE OF VARIABLE IN NT	
*	DMENT2 LNFSAVE,LNF	
	WRITE 6,'IDNF HAS BEEN CALLED'	
	LR 1,5	REG 1 HAS INX VALUE
	CALL NTSEARCH	REG 1 HAS NT ENTRY ADDRESS
	LR 10,1	
	USING NTREG,10	ACCESS NT ENTRY
	L 9,VSTOP	ACCESS TOP OF VS
	LA 9,VSWIDTH(,9)	SET UP FOR NEW ENTRY
	ST 9,VSTOP	RESTORE NEW VSTOP VALUE
	USING VSREG,9	ACCESS NEW VS ENTRY
	SR 2,2	ZERO OUT
	ST 2,VSTAG	TAG FIELD
	MVC VSVALUE(8),NTVALUE	PUSH VALUE TO VS
	MVC VSTYPE(1),NTTYPE	SET TYPE FLAG
	MVC INFTEMP(1),NTTAG	ACCESS TAG IN NT
	NI LNFTIMEP,B'00111111'	BLANK LOCAL AND MARK
	MVC VSTAG(1),LNFTIMEP	SET TAG
	CLI LNFTIMEP,TAGDT	CHECK FOR TAG DT (ARRAY)
	BNE LNFINISH	BRANCH IF NOT
	L 7,NTVALUE+4	GET DA POINTER
	A 7,MEMADDE	MAKE ABSOLUTE
	ST 7,VSVALUE+4	SET NT POINTER
	USING DESCRIPT,7	ACCESS DA
	L 6,DAFILL	LOAD WORD CONTAINING RC
	LA 6,1(,6)	INCREMENT BY 1
	ST 6,DAFIIL	RESTORE VALUE
	MVI VSTAG,TAGFDT	SET TAG TO FDT
LNFINISH	DMEX LNFSAVE	
LNFTIMEP	DS 4F	
	DROP 7,9,10,11,12	
	ITCRG	
	DS 10F	



```

TITLE 'TRANSFER OF CCNTROL'
ENTRY JMP
JMF      DS      0D
*
* OPERATOR      CFCODE IN REG 8
* JMF          17
* JMP0         18
* JMP1         19
*
REG 6 HAS K FCR ANY JUMP
DMENT JMFSAVE,JMP
L      10,ISTCP
USING LSREG,10
L      9,VSTOP
USING VSREG,9
L      3,VSVALUE+4
C      8,=F'18'
BL     JUMFCHNG
BE     JMP0
S      3,=F'1'
JMP0   LTR     3,3
BM     NCJUMP
JUMFCHNG L     5,LSREL
IR     1,5
AR     5,6
ST     5,LSREL
NOJUMP STM     0,15,JMTEMP
DMEX   JMFSAVE
JMTEMP DS      16F
DROP   9,10,11,12
ITORG
DS      10F

```

ACCESS TOP ENTRY OF LS

ACCESS TOP ENTRY OF VS  
LOAD WORD 2 OF VALUE  
CHECK FOR WHICH JMF TYPE  
OPERATOR IS UNCCNDITIONL  
OPERATOR IS JMP0  
SEE IF VALUE = 1  
TEST JMP0=0 AND JMP1=1  
DO NOT CHANGE REL FIELD  
ACCESS REL VALUE  
FOR TESTING ONLY  
CHANGE REL  
STORE NEW REL  
DUMP REGISTERS



## APPENDIX F.

SCALAR ARITHMETIC OPERATORS.

MDADIC LISTING.

GDF LISTING.



```

TITLE 'DYADIC AND MCNADIC ROUTINE'
MDADIC  CSFCT
        DMENT2 MDSAVE,MDADIC
*       REG 8 HAS CFCODE 21 FOR MONADIC, 22 FOR DYADIC
*       REG 6 HAS OFFEATOR, MCNADIC OR DIADIC
        SLL      6,2                DISPLACEMENT INTC TABLES
        L        10,VSTOP           ACCESS TOP OF VS
        S        10,=A(VSWIDTH)     DOWN TO SECOND VS ENTRY
        USING    VSREG,10           ACCESS SECOND TOP ENTRY
        C        8,=X'00000022'     CHECK IF DYADIC
        BE       MDYAD              IF SO, THEN BRANCH
*
*       DEALING WITH A MONADIC OPERATOR
        PUSH INSTRUCTION TO QS
        L        2,QSINSRT          SET UP FOR NEW ENTRY
        ST       2,QSTOP            UPDATE QSTOP VALUE
        USING    QSREG2,2           ACCESS NEW QS ENTRY
        LA       1,MCNTBLE(6)       ACCESS ADDRESS TABLE
        MVC      QSADDR2(3),1(1)    SET UP ROUTINE ADDRESS
        MVI      QSCF2,OPCP         SET OPCODE FOR MCNADIC
        MVI      13(10),TAGSGT      SET TAG TO SGT
        LA       2,QSWIDTH2(,2)     INCREMENT QSINSRT
        ST       2,QSINSRT          STORE AS NEW QSINSRT
        CII      VSTAG+VSWIDTH,TAGSGT CHECK IF A SEG DESC
        BE       MDFINISH           IF SO, THEN FINISHED
*
*       NOW MUST BUILD LS ENTRY AND CALL EM
MDMON1  L        9,LSTOP            ACCESS TOP OF LS
        LA       9,LSWIDTH(,9)     SET UP FOR NEW ENTRY
        ST       9,LSTOP            STORE NEW TOP VALUE
        USING    LSREG,9           ACCESS NEW ENTRY
        MVC      LSREL(4),=F'0'     SET REL=0
        MVC      LSCFG(4),QSTOP     SET POINTER TO QS SEGMENT
        MVC      LSDE(4),=F'4'      SET LENGTH TO 4 BYTES
        MVI      LSDE,B'10000000'   SET FLAGS
        B        MDFINISH           THEN FINISHED
*
*       DEALING WITH A DYADIC OPERATOR
MDYAD   CII      VSTAG,TAGSGT       CHECK IF SECOND SEG DESC
        BE       MDYAD2SD           IF SO, THEN ERANCH
*
*       NOW DEALING WITH A DYADIC, SECOND IS A SCALAR
        CII      VSTAG+VSWIDTH,TAGSGT CHECK IF TOP SEG DESC
        BE       MDYAD1SD           IF SO, THEN ERANCH
*
*       NOW DEALING WITH A DYADIC SCALAR SITUATION
        L        2,QSINSRT          SET UP NEW ENTRY
        ST       2,QSTOP            STORE NEW QSTOP VALUE
        USING    QSREG2,2           ACCESS NEW ENTRY
        LA       1,DYDTBLE(6)       ACCESS ROUTINE ADDRESS
        MVC      QSADDR2(3),1(1)    SET UP ROUTINE ADDRESS
        MVI      QSCF2,CPGOP        SET OPCODE FOR DYADIC
        LA       2,QSWIDTH2(,2)     UPDATE QSINSRT
        ST       2,QSINSRT          STORE NEW QSINSRT VALUE

```





```

      B      MDMCN1          CALL EM
*      NCW TCP IS SEG DESC AND SECOND IS SCALAR
*      MUST MOVE LAST QS SEG UP FOR NEW INSTRUCTION
MDYAD1SD LA      1,MD101      SET BRANCH ADDRESS
MD100    L       7,VSTAG      ACCESS TAG FIELD
        SIL      7,14         ISOLATE TYPE FLAG
        SRL      7,30         ISOLATE VALUE LENGTH
        LA       7,3(,6)
        SRL      7,2
        SIL      7,2          GO UP TO NEXT FULL WORD
        BR       1            RETURN
*      REG 7 NOW HAS LENGTH OF INSTRUCTION TO BE INSERTED
*      NOW MOVE QS SEG UP
MD101    L       9,QSINSFT     ACCESS END OF SEGMENT
        LR       5,9
        SR       5,7          5 HAS ADDRESS MOVED FROM
        L        4,VSVALUE+4+VSWIDTH SEG DESC POINTER
MDMOVE   EX      7,MDMVC       MOVE UP A SECTION OF CODE
        SR       9,7
        SR       5,7          DECREMENT INDICES
        CR       5,4          CHECK IF FINISHED
        BNH      MDINSERT
        B        MDMCVC
MDMVC    MVC     0(0,9),0(5)
*      INSERT THE INSTRUCTION IN THE QS
MDINSERT LA      1,MDUPENTR
        LR       2,4          2 HAS ADDRESS FOR INSERT
        LR       3,10         3 HAS ADDRESS INSERTED
        USING    VSREG,3      ACCESS VS ENTRY
MDINSRT1 CII     VSTYPE,B'00000010' CHECK LENGTH 4 BYTES
        BE       MDINLEN4     INTEGER OR REAL VALUE
        BH       MDINLEN8     LONG REAL VALUE
*      INSERT A BINARY VALUE OR A CHARACTER VALUE
        MVC     0(4,2),VSVALUE+4 MOVE IN OPERAND
        MVI     0(2),OPC1     SET OPCODE
        B       MDINSRTF      THEN FINISHED
*      INSERT AN INTEGER VALUE OR A REAL VALUE
MDINLEN4 MVI     0(2),OPC4     SET OPCODE
        MVC     4(4,2),VSVALUE+4 MOVE IN OPERAND
        B       MDINSRTF      THEN FINISHED
*      INSERT A LONG REAL VALUE
MDINLEN8 MVI     0(2),OPC8     SET OPCODE
        MVC     4(8,2),VSVALUE MOVE IN OPERAND
MDINSRTF EF      1            RETURN
*      NCW UPDATE PCINTERS
MDUPENTR L       9,QSINSRT
        DROP    3            DROP USING REGISTER
        L       5,QSTOP       ACCESS POINTERS
        AR      9,7
        AR      5,7

```



```

AR      4,7                UPDATE POINTERS
SI      9,QSINSRT
SI      5,QSTCP
ST      4,VSVALUE+4+VSWIDTH  STORE NEW SGT POINTER
MVI     VSTAG,TAGSGT        SET TAG TO SGT
B       MDGCPINS            BRANCH TO INSERT GOP INST
*
DEALING WITH SECCND ENTRY AS A SEG DESC
MDYAD2SD CLI  VSTAG+VSWIDTH,TAGSGT  CHECK TOP A SEG DESC
BE      MDYAD12S            IF SO, BRANCH,
*
DEALING WITH TCP A SCALAR AND SECOND A SEG DESC
L       2,QSINSRT           FOR LOAD SCALAR INSTRUCT
LR      3,10                ADDRESS OF INSERTEE
LA      3,VSWIDTH(,3)       LOCATION OF VS ENTRY
USING   VSREG,3             ACCESS TOP VS ENTRY
BAL     1,MDINSRT1          INSERT INSTRUCTION
*
NOW CHANGE VS ENTRY AND UPDATE PCINTER
MVI     VSTAG,TAGSGT        SET TAG TO SGT
ST      2,VSVALUE+4         SET POINTER TO QS SEGMENT
BAL     1,MD100             GET LENGTH OF INSTRUCTION
L       2,QSINSRT           UPDATE QSTOP
ST      2,QSTOP             STORE NEW QSTOP VALUE
AR      2,7                 UPDATE QSINSRT
ST      2,QSINSRT           STORE NEW QSINSRT VALUE
DROP    3                   DROP USING REGISTER
*
DEALING WITH 2 SEGMENT DESCRIPTORS
MDYAD12S SR    1,1           SET REG 1 FOR CCNFCRM
L       3,VSVALUE+4         GET SGT POINTER 2
L       2,0(3)              2 HAS DA POINTER SEG 2
L       4,VSVALUE+4+VSWIDTH GET SGT POINTER 1
L       3,0(4)              3 HAS DA POINTER SEG 1
CALL    CCNFORM             CHECK IF CONFORMABLE
*
NOW PUSH GOP INSTRUCTION
MDGCPINS L      5,QSINSRT    UPDATE QSTOP
ST      5,QSTCP             SET NEW QSTOP VALUE
USING   QSREG2,5           ACCESS NEW ENTRY
LA      1,DYDTBLE(6)        ACCESS ROUTINE ADDRESS
MVC     QSACDR2(3),1(1)     SET UP ROUTINE ADDRESS
MVI     QSOE2,CPGCP         SET OP CODE TO GOP
LA      5,QSWIDTH2(,5)      UPDATE QSINSRT
ST      5,QSINSRT
*
POP VALUE STACK
L       5,VSTOP             ACCESS VS
S       5,=A(VSWIDTH)       POP ENTRY
ST      5,VSTOP             RESTORE NEW VSTOP VALUE
MVI     1(5),TAGSGT         SET TAG TO SGT
MDFINISH DMEX  MDSAVE
MCNTBLE IS    20F
DYDTBLE DS    20F
DROP    2,3,9,10,11,12
LTORG

```



```

GDF      TITLE ' GENERAL DYADIC FORM SECTION'
CSECT
DMEN12  GDSAVE,GDF
STM     0,15,GDDISP
WRITE   6,'GDF WAS CALLED'
ST       6,GLOPER                STORE AFL OPERATOR #
*
CHECK   IF EITHER OPERAND IS A SCALAR VALUE
L       10,VSTOP                ACCESS TOP OF VS
USING   VSREG,10                ACCESS TOP ENTRY
CLL     VSTAG,TAGST             CHECK IF SCALAR
BE      GDDYADIC                IF YES, TREAT AS DYADIC
S       10,=A(VSWIDTH)          DECREMENT TO NEXT ENTRY
CLL     VSTAG,TAGST             CHECK IF SCALAR
BE      GDDYADIC                IF YES, TREAT AS DYADIC
*
CHECK   IF HAVE TO EVALUATE RHO IMMEDIATELY
CLL     VSTAG,TAGDT             CHECK IF AN ARRAY
BE      GDGCFINS                IF YES, PUSH GOF INST
*
NOW MUST CALL EM TO EVALUATE RHO
*
FIRST, MUST INSERT SOME INSTRUCTIONS IN QS
L       8,QSTOP                 ACCESS TOP OF QS
LA      8,32(,8)                UPDATE FOR ADDED LENGTH
ST      8,QSTOP                 RESTORE
L       8,QSINSFT               ACCESS END OF QS
LR      9,8                     FOR LATER USE
LA      8,32(,8)                UPDATE
ST      8,QSINSFT              RESTORE
L       7,=F'32'                LENGTH OF MOVE
L       6,=F'31'                FOR EX STATEMENT
LR      5,9                     REG 5 IS INDEX
SR      5,7                     DECREMENT INDEX
L       4,VSVALUE+4             GET QS POINTER
LA      3,GDINSFT1              FOR BRANCHING
S       10,=A(VSWIDTH)          DOWN NEXT ENTRY
*
MCVE    UP A CHUNK OF CODE
GDMOVE  EX      6,GDMVC          MOVE CHUNK OF CCDE
SR      9,7                     DECREMENT DESTINATION
SR      5,7                     DECREMENT ORIGIN
CR      5,4                     CHECK IF DONE
BH      GDMCVC                  IF NOT, CONTINUE
BR      3                       OTHERWISE, FINISHED
GDMVC   MVC     0(0,9),0(5)
*
NCW MUST INSERT SCME ENTRIES IN QS
GDINSRT1 L      1,VSVALUE        GET IS POINTER
*
ALLOCATE CREATES DA AND ARRAY SPACE FOR TEMP RESULT
*
POINTER CCMES VACK IN REG 1 AND MASK IN REG 2
CALL    ALLCCATE
LA      4,4(,4)                 UP FOR IA INSTRUCTION
USING   QSREG5,4                ACCESS NEW INSTRUCTION
MVI     QSOF5,OPIA              SET OPCODE TO IA

```





```

MVC      QSLINK5(3),=X'000000'  ZERO LINK FIELD
ST       1,QSADDR5              SET DA POINTER
ST       2,QSMASK5              SET MASK
*
THE IA TEMP INSTRUCTION IS CREATED
LA       4,QSWIDTH5(,4)         UP FOR ASGN INSTRUCTION
MVC      0(4),=V(EMASGN)        ADDRESS OF ASSIGN ROUTINE
MVI      0(4),OPOP              SET OPCODE
*
THE ASGN INSTRUCTION IS CREATED
LA       4,QSWIDTH2(,4)         UP FOR IFA TEMP
MVI      QSOP5,OPIFA           SET OPCODE
MVC      QSLINK5(3),=X'000000'  ZERO LINK FIELD
ST       1,QSADDR5              SET DA POINTER
ST       2,QSMASK5              SET MASK
*
THE IFA TEMP INSTRUCTION HAS BEEN CREATED
L        9,VSVALUE+4            TOP OF NEXT CHUNK
ST       4,VSVALUE+4            UPDATE SGT PCOUNTER
LR       8,4                    FOR LATER USE
L        7,=F'4'                LENGTH OF MOVE
L        6,=F'3'                FOR EX STATEMENT
LR       5,9                    REG 5 IS INDEX
SR       5,7                    DECREMENT INDEX
L        4,VSVALUE+4            GET QS POINTER
LA       3,GDINSRT2            FOR BRANCHING
B        GDMCVE                 MOVE SECTION OF CODE
*
NOW MUST INSERT JMP INSTRUCTION
GDINSRT2 LR      9,4            NEED DUPLICATE
LR       7,4                    AND ANOTHER
SR       8,9                    LENGTH OF JUMP
ST       8,0(,4)                SET LENGTH OF JUMP
MVI      0(4),OPJMP            SET OPCODE
*
NOW MUST CREATE LS ENTRY TO CALL EM
L        8,LSTCF                ACCESS IS
LA       8,LSWIDTE(,8)          ALLOCATE NEW ENTRY
USING    LSREG,8                ACCESS NEW ENTRY
SR       2,2
ST       2,ISREL                ZERO REL FIELD
LA       4,4(,4)                UP 1 INSTRUCTION
ST       4,LSORG                SET ORG FIELD
MVI      LSDE,B'10000000'       SET DE FLAG CN
SR       7,4                    LENGTH OF SEGMENT
ST       7,LSDE                SET LENGTH FIELD
ST       2,LSQP                ZERC QP FIELD
STM      0,15,GDTEMPEV          STORE REGISTERS
LA       3,GDTEMRET            SET RETURN ADDRESS
ST       3,TEMPRET
B        GDFINISH                PASS CCNTROL TO EM
GDTEMRET LM      0,15,GDTEMPEV  RESTORE REGISTERS
SR       2,2
ST       2,TEMPRET              ZERO TEMPRET
*
NOW FINISHED WITH IMMEDIATE EVALUATION

```





```

*          NOW MUST PUSH A GOP OPERATOR INSTRUCTION
GDGOPINS  L      9, QSINSFT          ACCESS TOP OF QS
          USING  QSREG2, 9          ACCESS NDW ENTRY
          L      6, GDOPER          GET APL OPERATOR
          SIL    6, 2              INTO BYTES
          S      6, =F'4'          BACK 1 WORD
          A      6, =A (GDOPTBL)    DISPLACEMENT INTO TABLE
          MVC    QSOF2(4), 0(6)      LOAD ADDRESS EM ROUTINE
          MVI    QSOF2, CFGOP        SET OPCODE
          LA     9, QSWIDTH2(, 9)    UPDATE QSINSRT
          ST     9, QSINSRT          RESTORE
          B      GDFINISH           NOW FINISHED

*          NOW MUST HANDLE THE SCALAR OPERANDS
GDDYADIC  BCTR   8, 0              CHANGE OPCODE TO DYADIC
          L      15, =V(MDADIC)     ADDRESS OF DYADIC ROUTINE
          BALR   14, 15

GDFINISH  DMEX   GDSAVE
GDOPTBL   LS     22F
GDTEMPEV  DS     16F
GDDISP    DS     16F
GDOPER    DS     1F

```



## APPENDIX G.

SELECTION OPERATORS.

## SELECT LISTING.



```

TITLE 'SELECT SECTION'
SELECT CSECT
      DMENT2 SESAVE,SELECT
      WRITE 6,'SELECT HAS BEEN CALLED'
      STM 0,15,SEDUMP
      ST 8,SETEMP
*      IF TOP SEGMENT IS NOT AN ARRAY, EVALUATE IT
      L 10,VSTOP GET TOP OF VS
      USING VSREG,10 ACCESS TOP ENTRY
      CII VSEXP,B'00000011' CHECK EXPRESSION FOR ARRAY
      BE SE2TCP IF YES, CHECK SECOND ENTRY
      CALL IMMED1 EVALUATE TOP ENTRY
SE2TOP S 10,=A(VSWIDTH) DOWN TO SECOND ENTRY
      CII VSBET,B'00000001' CHECK IF BEATABLE
      BZ SEACCD A IF YES, START BEATING
      CALL IMMED2 EVALUATE SECOND TOP ENTRY
*      NOW MUST START BEATING DAS IN SECOND SEGMENT
SEACCD A L 9,VSVALUE+4 GET QS PCINTER
      L 7,VSVALUE+VSWIDTH+4 GET NEXT QS POINTER
*      REG 9 USED AS INDEX THROUGH QS
SECHQIFA CLI 0(9),OPIFA CHECK IF IFA INSTRUCTION
      BE SEBEAT YES?, BRANCH FOR BEATING
      L 8,0(,9) LENGTH OF INSTRUCTION
      SRL 8,30 ISOLATE IT
      AR 9,8 INCREMENT INDEX
      CR 7,9 CHECK IF FINISHED SEGMENT
      BH SECHQIFA RETURN FOR NEXT DA
      B SEFINISH OTHERWISE, FINISHED
SEIFALOP L 8,0(,9) LENGTH OF INSTRUCTION
      B SEFINISH OTHERWISE, FINISHED
*      REG 9 POINTS TO QS INSTRUCTION WHICH POINTS TO A DA
      USING QSREG5,9 ACCESS THAT INSTRUCTION
      L 6,QSADDR5 REG 6 HAS DA POINTER
      USING DESCRIPT,6 ACCESS DA
      CLC DAREFCNT(2),=F'1' CHECK IF RS IS CNE
      BE SEBEAT THEN PROCEED WITH BEATING
*      RC>1, THEREFORE MUST COPY DA AND THEN BEAT
      L 2,4(,9) DA POINTER
      CALL COPYDA THE DA IS COPIED
*      NOW MUST CHOOSE TYPE OF BEATING OPERATION
SEBEAT L 2,VSVALUE+VSWIDTH+4 QSPCINTER FOR IHC
      LR 3,6 DA POINTER
      L 5,SETEMP ACCESS THE OPCCDE
      C 8,=F'9' CHECK FOR DROP
      BH SETAKE BRANCH FOR TAKE
      BE SEDROP BRANCH FOR DRCE
      C 8,=F'10' CHECK IF TRANSECSE
      BE SETFAN BRANCH IF TRANSECSE
      CALL SELREV APPLY REVERSAL OPERATION

```



	B	SEIFALOP	RETURN FOR NEXT LA
SETAKE	CALL	SEITAKE	APPLY TAKE PROCESS
	B	SEIFALOP	RETURN FOR NEXT LA
SEDROP	CALL	SEIDROP	APPLU DROP PROCESS
	B	SEIFALOP	RETURN FOR NEXT LA
SETRAN	CALL	SELTRANS	APPLY TRANSPCSE PROCESS
	B	SEIFALOP	RETURN FOR NEXT LA
SEFINISH	DMEX	SESAVE	
SETEMP	DS	1F	
SEDUMP	DS	16F	





## APPENDIX H.

PROGRAM CONSTANTS.

MACROS.

EQUATES.

APLMCONS.

QSCPCODES.

STACK REGISTERS.



```

TITLE 'LINKAGE CCNVENTICN MACROS'
MACRO
&LABEL DMENT2 &SAVE,&NAME
&LABEL STM 14,12,12(13)
LR 12,15
LR 15,13
USING &NAME,12
USING APLMCCNS,11
LA 13,&SAVE
ST 15,&SAVE+4
ST 13,8(,15)
MEND
MACRO
&LABEL DMENT &SAVE,&NAME
&LABEL STM 14,12,12(13)
LR 12,15
LR 11,13
USING &NAME,12
USING AFLMCCNS,11
LA 13,&SAVE
ST 11,&SAVE+4
ST 13,8(,11)
MEND
MACRO
&LABEL CMEX &SAVE
&LABEL L 13,&SAVE+4
IM 14,12,12(13)
BR 14
CNOB 0,4
ITORG
&SAVE DS 20F
MEND
TITLE 'MACRC DEFINITIONS USED IN EMACHINE'
MACRO
&LABEL CALLREC &R
&LABEL L 14,BCTTECOL
L 15,=V(&R)
BALR 10,15
MEND
SPACE 2
MACRO
&LABEL CALLRENT &RCUT
* CALL OF A ROUTINE THAT IS RE-ENTERANT
&LABEL L 15,=V(&ROUT)
BALR 10,15
MEND
SPACE 2
MACRC
&LABEL CALL0 &R

```



```

I      1,=F'-291'
L      15,=V(&R)
EALR   14,15
MEND
SPACE 2
MACFO
&LAB   QSOPS &A,&I=256,&I=0,&ST=1
.* I INDICATES INSTRUCTION AN INITIALIZATION INSTRUCTION
.* SUCH AS IFA,IA,ETC. ST INDICATES INSTRUCTION SHOULD BE
.* SKIPPED WHEN THE IS STRUCTURE BIT IS ON.
OP&A   EQU    (64*&I)+(*-QSOFTABL)/4
&LAB   DC     1B'&I.&ST.000000'
        DC     V13(&A)
        MEND
        SPACE 2
        MACRC
&LABEL RECURENT
        USING *,11
        USING APLMCCNS,13
.* THIS ROUTINE SHCUID EE ENTERED ONLY BY CALLREC
&LABEL STM    0,12,8(14)
        LR     11,15
        MVC    0(8,14),SAVEARR
        LA     12,60(,14)
        ST     12,BCTTECCL
        S      14,TCFFCCL
        A      14,=A(140)
        BNP    *+6
        DC     1H'0'
        MEND
        SPACE 2
        MACRC
&LABEL RECURET
&LABEL C      12,ECTTECOL
        BE     RET&SYSNDX
        LR     1,12
        S      1,=A(60)
        CALLREC EFASEARR
        LM     0,12,8(1)
        BR     10
RET&SYSNDX S  12,=A(60)
        ST     12,ECTTECCL
        LM     0,12,8(12)
        BR     10
        DROP   11,13
        MEND
        SPACE 2
        MACRC
&LABEL RENTENT
* RE-ENTRANT ENTRY TO ROUTINE - BUT NOT RECURSIVE ENTRY

```



```

.* REG 10 - CCNTAINS ADDRESS TO RETURN CALLING PROGRAM.
.*          I.E. PALR 10,15
.* REG 11 - BASE REGISTER FOR PROGRAM CALLED.
.* REG 12 - ADDRESS CF SAVE AREA FOR EMACHINE ROUTINES.  IT
.*          PRVIDES FOR RE-ENTERANT AND RE-ENTERANT,
.*          RECURSIVE CALLS.
.* REG 13 - ADDRESS OF SAVE AREA WITH OS/360 LINKAGE
.*          CCNVENTICNS.  MAY OR MAY NOT BE RE-ENTRANT.
.*          CAN NCT PROVIDE FOR RECURSIVE CALIS.  ALSO
.*          ADDRESS OF APLM CCNSTANTS.
.* REG 14 - REGISTER USED TO CALCULATE TEMPORARY VALUES.
.* REG 15 - ENTRY PCINT ADDRESS OF CALLED ROUTINE.
.*          THEN SIMIIAF TO REG 14
                USING *,11          ADDRESSABILITY FCR ROUTINE
                USING APLMCONS,13    CONSTANT ADDRESSIBIITY
&LABEL STM 10,11,0(12)             REGISTERS SAVE
                LR 11,15             CALLED ROUTINE BASE REG
*                                     INITIALIZED
                LA 12,8(,12)         NEW SAVE AREA ADDRESS
                MEND
                SPACE 2
                MACRO
&LABEL RENTRET
* RECURSIVE RETURN
.* THIS MACRO IS USED TC RETURN CCNTROL WHEN ROUTINE USED
.* RENTENT TO DO THE ENTRY
&LABEL S 12,=A(8)                  ADDRESS OF BACK SAVE AREA
                LM 10,11,0(12)      CALLING BASE, RETURN PCINT
                BR 10                RETURN
                DROP 11,13
                MEND
                MACRO
&LABEL VSERR &T
                AIF ('&T' EQ 'FULL').FULL
                MNOTE 4,'&T IS AN UNRECOGNIZABLE OPERAND'
                MEXIT
.* FULL
&LABEL ANOP
                BNH *+6
                DC 1H'0'
                MEND

```









TAGFMT	EQU	4	VALUE OF TAG FOR FUNCTION MARK
TAGFT	EQU	5	VALUE FUNCTION DESCRIPTOR FNTR.
TAGAT	EQU	6	VALUE OF TAG ENCODED M-ADDRESS
TAGNPT	EQU	7	VALUE OF TAG FOR NAME POINTER
TAGRT	EQU	8	VALUE OF TAG FOR REDUCT ACCUM
TAGSGT	EQU	9	VALUE OF TAG SEGMENT DESCRIPTOR
TAGST	EQU	10	VALUE OF TAG FOR SCALAR VALUE
TAGUT	EQU	11	VALUE OF TAG UNDEFINED VALUE
TAGNLT	EQU	12	VALUE OF TAG FOR END ICB BLOCK
TAGNT	EQU	13	VALUE OF TAG FOR ICB BLOCK
TAGQLT	EQU	14	VALUE OF TAG FOR END ICB BLOCK
TAGQT	EQU	15	VALUE OF TAG FOR ICB BLOCK
SPACE 4			
LSDEON	EQU	B'10000000'	MASK BIT TO SET, TEST DE
EMAC	EQU	ISDECN	
LSISON	EQU	B'01000000'	MASK BIT TO SET, OR TEST IS
LSFNEN	EQU	B'00100000'	MASK BIT TO SET, CR TEST FN
LSNWTN	EQU	B'00010000'	MASK BIT TO SET, TEST NWT
NEWITON	EQU	B'10000000'	SET, TEST NEWIT
NEWITOFF	EQU	B'01111111'	MASK BIT TO SET, CR TEST
*			IF, NEWIT IS OFF
STPTGON	EQU	B'01000000'	MASK BIT TO SET, OR TEST
*			IF, STEPTOG IS ON
STPTGOFF	EQU	B'10111111'	MASK TO TURN STEPTOG OFF
LSTRTON	EQU	B'00001000'	STRUCTURE BIT IS ON IN LS
LSTRTOFF	EQU	B'11110111'	STRUCTURE BIT OFF IN LS
ISTRTON	EQU	B'00010000'	STRUCTURE BIT ON IN IS
ISDIRON	EQU	B'10000000'	MASK BIT TO SET, CR TEST
*			IF DIRECTION IS ON
ISCHON	EQU	B'01000000'	MASK BIT TO SET, OR TEST
*			IF, CHANGE IS ON.
ISMRKON	EQU	B'00100000'	MASK BIT TO SET, CR TEST
*			IF MRK IS ON
ISDIROFF	EQU	B'01111111'	TURN DIRECTION BIT OFF
X3CN	EQU	B'10000000'	MASK BIT TO TEST VALUE X3
ISCHOFF	EQU	B'10111111'	TURN CHANGE BIT OFF IN IS
ISBASON	EQU	B'00010000'	



```

      TITLE 'AFIMCONS - A DESCRIPTION OF ADDR. ETC IN EM'
AFIMCONS DSECT
PROGSAVE DC      9C'PRCGSAVE'  18 WORD SAVE AREA. SHOULD BE ON
      SPACE 3                      DOUBLE WORD BOUNDARY
* ADDRESS CONSTANTS IN AFIM
MEMADDR DC      A(MEMORY)      THE ADDRESS OF LOCATION ZERO
ARRAVAIL DC     1F'0'          FIRST NODE IN ARRAY AVAIL LIST
      DC     2A(ARRAVAIL)      NODE LOOKS LIKE ARRAY OF 0
* LENGTH - THUS CAN NOT BE ALLOCATED.
IAAVAIL DC     1F'0'          FIRST NODE IN DESCRIPTOR ARRAY
      DC     2A(IAAVAIL)      AVAILABILITY LIST. NODE
* APPEARS TO HAVE ZERO LENGTH - THUS CAN NOT BE
* ALLOCATED.
BOTTPool DC     A(4)          ABSOLUTE ADDRESS OF THE LOW END
* CF THE PCOL - ONE WORD PAS MEMEND. LOW END
* HAS LOW VALUE ADDRESS.
TOPPool DC     A(X'FFFFFF')    ABSOLUTE ADDRESS OF THE HIGH
* NEND OF THE PCOL. ADDRESS LIKELY DOES
* NOT HAVE VALUE INDICATED.
ISMARK DC     A(ISBASE-ISWIDTH) ABSOLUTE ADDRESS OF FIRST
* (CICSEST TO THE TOP OF THE STACK)
* ENTRY IN IS TO HAVE ISMRK BIT ON.
QSCNTRL DC     1F'0'          IDENTIFIES INSTRUCTION IN
* QS THAT CAN PUSH ENTRIES TO THE
* IS FOR INDEXING STRUCTURES.
      SPACE 4
DBI DS      1D              TEMPORARY LOCATION - USED TO
* CERTAIN BOUNDARY ALIGNMENT. EXAMPLE, DOUBLE
* WORD OPERANDS IN VSREG
SAVEARR DC     C'SAVEARR '
      SPACE 4
* THE FOLLOWING ACCNS OBEY THE FOLLOWING CONVENTIONS:
* (1) THE "BASE" ADDRESS POINTS AT AN ENTRY BEFORE THE
* FIRST ENTRY IN THE STACK. FOR EXAMPLE, THE FIRST
* ENTRY TO BE PLACED IN THE VS IS AT "VSBASE + VSWIDTH"
* (2) THE "SCAN" ADDRESS POINTS AT ANY ENTRY THAT IS
* CURRENTLY IN THE STACK. FOR EXAMPLE, THE CURRENT
* INSTRUCTION POINTER USED IN QSREG IS QSSCAN
* (3) THE "TOP" ADDRESS POINTS AT THE LAST ENTRY TO BE
* PLACED IN A STACK, THIS IMPLIES THAT "TOP" EQUALS "END"
* WHEN THE STACK IS FULL, AND THAT "TOP" EQUALS "BASE"
* IS AN ERROR I.E. WHEN THE STACK IS EMPTY "TOP"
* EQUALS "BASE" + "WIDTH"
* (4) THE "END" ADDRESS IS THE UPPER BOUND - ACTUALLY
* HIGHEST ADDRESS - AT WHICH ENTRIES MAY BE PLACED.
*
ISEND DS      1A
ISTOP DS      1A
ISSCAN DS      1A

```











DATYPE	DS	0F11
DALEN	DS	1F
DAFILL	DS	1H
DAREFCNT	DS	1H
DAVBASE	DS	1F
LAABASE	DS	1F
DARANK	IS	1F
DADIMEN	DS	1F
DACEI	DS	1F



## TITLE 'E-MACHINE INSTRUCTION AND FORMATS'

CNOP 0,4

QSOPHTABL CS 0H

\*

\* |OP:XX|XX:XX| TYPE 1

\*

\* OP -OPERATION CODE

\* XX -UNUSED BYTES

QSOPS CY,I=1

QSOPS IRP,I=1,I=1

QSOPS LVE,L=1

QSOPS MIT,I=1

QSOPS NIL,I=1

QSOPS ORG,L=1

QSOPS PCFVS,I=1

QSOPS RPT,I=1

QSOPS VXC,I=1

SPACE 3

\*

\* |OP:&lt;A:DD:R&gt;| TYPE 2

\*

\* ADDR-24 BIT DISPLACEMENT OR ABSOLUTE ADDRESS

QSCPS CAS,I=1

QSOPS DUP,I=1

QSOPS GCF,I=1

QSOPS IRD,L=1,I=1

QSCPS IXL,L=1,I=1

QSOPS JMF,I=1

QSOPS JNO,I=1

QSOPS JN1,I=1

QSCPS JO,I=1

QSOPS J1,I=1

QSOPS IX1,I=1

QSOPS LX2,I=1

QSCPS CP,I=1

QSOPS RED,I=1

QSOPS SX1,L=1

QSOPS SX2,L=1

QSCPS XC,I=1

QSOPS XI,L=1

QSOPS XS,I=1

SPACE 3

\*

\* |OP:XX:TY:VL| TYPE 3

\*

\* TY -CNE BYTE DENOTING DATA TYPE

\* VL -CNE BIT OR BYTE OF LOGICAL OR CHARACTER

QSOPS S1,I=1

EJECT



```

*
* |OP:XX:TY:XX| | V:AI|UE:2 | TYPE 4
*
*          VALUE2-FULL WORD INTEGER OR REAL DATA
*          QSOPS S4,I=2
*          SPACE 3
*
* |OP:<O|RG:>>| |MD:<L|EN:>>| TYPE 5
*
*          ORG    -24 BIT DISPLACEMENT
*          LEN    -24 BIT LENGTH
*          MD     -LEFTMOST BIT OF BYTE IS
*          MODE FOR SGV
*          -LEFTMOST BIT IS DIRECTION FOR IJ
*          QSOPS IJ,I=2,I=1
*          QSOPS SGV,I=2
*          EJECT
*
* |OP:XX:TY:XX| |<V:AI|UE:3>| |<V:AI|UE:3>| TYPE 6
*
*          VALUE3-64 BIT REAL DATA
*          QSOPS S8,I=3
*          SPACE 3
*
* |OP:<M|AS:K>| |XX:<A:DD:R>| |XX:LN|KD:SP| TYPE 7
*
*          MASK    - 24 BIT ARRAY ACCESS
*          MASK
*          LNKDSP-24 BIT LINK
*          DISPLACEMENT VALUE
*          QSOPS A,L=3,ST=0
*          QSOPS FA,I=3,ST=0
*          QSOPS IA,I=3,I=1
*          QSOPS IFA,L=3,I=1
*          SPACE 3
*
* |OP:<O|RG:>>| |MD:<I|EN:>>| |XX:LN|KD:SP| TYPE 8
*
*          QSOPS ISC,I=3,I=1
*          QSOPS J,L=3
*          QSOPS SC,I=3
*          QSOPS SG,I=3

```



## TITLE 'E-MACHINE REGISTER FORMATS'

AFLMREGS DSECT

\*

\* |XX:&lt;C:TR:&gt;&gt;| |FL:&lt;M|AX:&gt;&gt;|

\*

ISREG DSECT

ISCTR DS 1F

ISDIR DS 0B

ISCH DS 0B

ISMRK DS 0B

ISTRT DS 0B

ISBASM RK DS 1B

ISMAX DS 1FI3

ISWIDTH EQU \*-ISREG

SPACE 3

\*

\* |XX:&lt;R|FL:&gt;&gt;| |XX:&lt;C|RG:&gt;&gt;| |FL:&lt;L|FN:&gt;&gt;| |XX:&lt;&lt;|QP:&gt;&gt;|

\*

\* REL-24 BIT COUNTER

\* CRG-24 BIT DISPLACEMENT

\* FI -FLAGS D/E, IS, FN, NWT

\* LEN-24 BIT LENGTH

\* QP -24 BIT DISPLACEMENT

LSREG DSECT

LSREL DS 1F

LSORG DS 1F

LSDE DS 0B

LSIS DS 0B

LSFN DS 0B

LSNWT DS 0B

LSTRT DS 1B

LSLEN DS 1FI3

LSQP DS 1F

LSWIDTH EQU \*-LSREG

SPACE 3

NTREG DSECT

NTINX DS 1H

NTTYPE DS 1FL1

NTLOCAL DS 0B

NTMARK DS 0B

NTTAG DS 1B

NTVALUE DS 2F

NTWIDTH EQU \*-NTREG

SPACE 3

QSREG1 DSECT

QSOP1 DS FL1 OP CODE

DS FI3 UNUSED

QSWIDTH1 EQU \*-QSREG1

SPACE 3





QSREG2	DSECT		
QSCP2	DS	FI1	OP CODE
QSADDR2	DS	FI3	24 BIT ADDRESS
QSWIDTH2	EQU	*-QSREG2	
	SPACE	3	
QSREG3	DSECT		
QSCP3	DS	FI1	OP CODE
QSTYPE3	DS	FI1	TYPE OF VALUE
QSVALUE3	DS	FI1	VALUE - BOOLEAN OR CHARACTER
	DS	FI1	UNUSED
QSWIDTH3	EQU	*-QSREG3	
	SPACE	3	
QSREG4	DSECT		
QSCP4	DS	FI1	OP CODE
QSTYPE4	DS	FI1	TYPE OF VALUE
	DS	FI2	UNUSED
QSVALUE4	DS	F	VALUE - INTEGER, REAL, OR STRUCTURE
QSWIDTH4	EQU	*-QSREG4	
	SPACE	3	
QSREG5	DSECT		
QSCP5	DS	FI1	OP CODE
QSLINK5	DS	1FI3	LINK TO HIGHER LEVEL FA IN STRUCTURE
QSADDR5	DS	1F	D.A. ADDRESS
QSMASK5	DS	1F	32 BIT ACCESS MASK
QSWIDTH5	EQU	*-QSREG5	
	SPACE	3	
QSREG6	DSECT		
QSCP6	DS	FI1	OP CODE
QSTYPE6	DS	FI1	TYPE OF VALUE
	DS	FI2	UNUSED
QSVALUE6	DS	2F	VALUE - LONG REAL
QSWIDTH6	EQU	*-QSREG6	
	SPACE	3	
QSREG7	DSECT		
QSCP7	DS	FI1	OP CODE
QSTYPE7	DS	1FI1	
QSLINK7	DS	1FI2	ICB LINK
QSLLEN7	DS	1FI1	LENGTH IN BYTES OF TYPE
QSVBASE7	DS	1FI3	ADDRESS OF DATA
QSSUM7	DS	1F	INDEX TO CURRENT ELEMENT
QSWIDTH7	EQU	*-QSREG7	
	SPACE	3	
QSREG8	DSECT		
QSCP8	DS	FI1	
QX1	DS	FI3	
QX3	DS	FI1	
QX2	DS	FI3	
QSLINK8	EQU	*-QSREG8	
	SPACE	3	
QSICB	DSECT		



```

QSTAG      DS      FL1
QSQ1       DS      FL3
QSQ2       DS      F
QSINX      DS      F
QSICBWID   EQU     *-QSICB
           SPACE 3

```

```

*
```

```

*   :TG:XX:TY:VL|   |<V:AL|UE:>>|   |<<:VA|LU:E>|

```

```

*
```

```

           SPACE 3
SECDREG    DSECT
SDLEN      DS      1F
SERC       DS      1H
SDFILLER   DS      1H
SDFISR     DS      1FL1
SDEASE     DS      3FL1
SDORG      DS      1FL1
SDFLEN     DS      3FL1
SDFPARS    DS      1H
SDFLCLS    DS      1H
SDPARNMS   DS      1F
           SPACE 3
VSREG      DSECT
VSTAG      DS      1FL1
VSTYPE     DS      1FL1
VSEXP      DS      1FL1
VSBET      DS      1FL1
VSVALUE    DS      2F
VSWIDTH    EQU     *-VSREG

```



## REFERENCES.

- Abrams, P. S. (1966). An Interpreter for "Iverson Notation." Report No. CS47, Computer Science Department, Stanford University (August).
- Abrams, P. S. (1970). An APL Machine. Computer Science Department, Stanford University (February).
- Berry, P. (1969). APL 360 Primer. Form No. C20-1702-0, International Business Machines Corp., White Plains, New York.
- Breed, L. M. And Lathwell, R. H. (1968). The Implementation of APL 360. Interactive Systems for Applied Mathematics, Academic Press, New York.
- Collins, G. e. (1965). REFCO III, A Reference Count List Processing System for the IBM 7079. Research Report No. RC-1436, IBM Research Division, Yorkton Heights, New York.
- Falkoff, A. D. And Iverson, K. E. (1968). APL 360 User's Manual. International Business Machines Corp., Yorkton Heights, New York (August).
- Knuth, D. E. (1968). The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison Wesley, Reading, Massachusetts.
- Pakin, S. (1968). APL 360 Reference Manual, Science Research Association Inc., Chicago, Illinois.











**B30029**